Integration of Programming and Learning in a Control Language for Autonomous Robots Performing Everyday Activities

Alexandra Kirsch

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

Integration of Programming and Learning in a Control Language for Autonomous Robots Performing Everyday Activities

Alexandra Kirsch

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. Helmut Seidl

Prüfer der Dissertation:

1. Univ.-Prof. Michael Beetz, Ph.D.
2. Prof. Rachid Alami,
   LAAS/CNRS, Toulouse/Frankreich

Die Dissertation wurde am 07.08.2007 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 07.01.2008 angenommen.

Abstract

Robots performing complex tasks in changing, everyday environments and required to improve with experience must continually monitor the way they execute their routines and revise them if necessary. Existing approaches, which use either monolithic or isolated, nonrecurring learning processes, cannot sufficiently focus their learning processes to satisfy these requirements. To meet this challenge we propose to make learning an integral part of the control program by providing a control language that includes constructs for specifying and executing learning problems.

Our Robot Learning Language (RoLL) makes learning tasks executable within the control program. It allows for the specification of complete learning processes including the acquisition of experience, the execution of learning algorithms and the integration of learning results into the program. RoLL is built upon the concept of experience, which is a learning task specific symbolic summary of a problem solving episode. This means that experiences do not only record the observed data, but also include the robot's intentions and the perceived execution context. The experience acquisition in RoLL is designed in a way that experiences can be defined outside the primary control program, using hybrid automata as a tool for declaratively specifying experience and anchoring it to the program. The rich experience concept enables convenient abstraction and an economic use of experiences. RoLL's design allows the inclusion of arbitrary experience-based learning algorithms. Upon the completion of the learning process RoLL automatically integrates the learned function into the control program without interrupting program execution.

RoLL enables the plug-and-play addition of new learning problems and keeps the control program modular and transparent. RoLL's control structures make learning an integral part of the control program and can serve as a powerful implementational platform for comprehensive learning approaches such as developmental, life-long and imitation learning.

Zusammenfassung

Roboter, die komplexe Aufgaben in dynamischen, alltäglichen Umgebungen lösen und die sich durch Erfahrung verbessern sollen, müssen ständig überwachen wie sie ihre Routinen ausführen und diese wenn nötig korrigieren. Existierende Ansätze, die entweder monolitische oder isolierte, einmalige Lernprozesse benutzen, können ihre Lernprozesse nicht genügend fokussieren um diesen Anforderungen zu genügen. Um dieser Herausforderung gerecht zu werden, schlagen wir vor Lernen zu einem integralen Teil des Kontrollprogramms zu machen, indem wir eine Kontrollsprache bereitstellen, die Konstrukte zum Spezifizieren und Ausführen von Lernproblemen enthält.

Unsere Sprache RoLL (Robot Learning Language) macht Lernprobleme im Kontrollprogramm ausführbar. Es ermöglicht die Spezifikation von kompletten Lernprozessen einschlieẞlich der Beschaffung von Erfahrungen, der Ausführung von Lernalgorithmen und der Integration des Lernergebnisses in das Programm. RoLL baut auf dem Konzept der Erfahrung auf, die eine lernproblemspezifische symbolische Zusammenfassung einer Problemlösungsepisode ist. Dies bedeutet, dass Erfahrungen nicht nur eine Aufzeichnung der beobachteten Daten sind, sondern auch die Absichten des Roboters und den wahrgenommenen Ausführungskontext einschlieẞen. Der Erfahrungserwerb in RoLL is so entworfen, dass Erfahrungen auẞerhalb des eigentlichen Kontrollprogramms definiert werden können, wobei hybride Automaten als Werkzeug für die deklarative Spezifikation von Erfahrungen und deren Verankerung im Kontrollprogramm benutzt werden. Das ausdrucksstarke Erfahrungskonzept ermöglicht eine komfortable Abstraktion und ökonomische Verwendung von Erfahrungen. Das Konzept von RoLL erlaubt die Einbindung von beliebigen erfahrungsbasierten Lernalgorithmen. Sobald ein Lernprozess beendet ist, integriert RoLL die gelernte Funktion automatisch in das Kontrollprogramm ohne die Programmausführung zu unterbrechen.

Mit RoLL können dem Programm jederzeit neue Lernprobleme hinzugefügt werden, wobei der Code modular und verständlich bleibt. Die Kontrollstrukturen von RoLL machen das Lernen zu einem integralen Bestandteil des Kontrollprogramms und sie können als mächtige Implementierungsplattform für umfassende Lernansätze wie Entwicklungslernen (developmental learning), lebenslanges Lernen (life-long learning) und Lernen durch Imitation dienen.

Contents

Chapter 1

Introduction

> Tell me, and I will forget.
> Show me, and I may remember.
> Involve me, and I will understand.
>
> Confucius around 450 BC

1.1 Motivation

With rapid progress in hardware development, researchers attempt more and more to bring robotic applications into human working and living environments. However, the transition from well-defined industrial robot environments to complex, dynamic, and uncertain domains is extremely hard. Most researchers agree that machines working in human environments must — like humans — adapt their behavior to changing situations and enhance their performance by learning.

In recent years, machine learning has contributed a lot to the success of hitherto unimaginable robotic systems operating in real-world scenarios (Thrun et al. 2006). Learning enables robots to adapt to specific environments and frees programmers from tedious and error-prone parameter tuning. Despite its momentousness, in most robotic applications learning is seen as an external resource of the system rather than integrated component. Once the learning is done, the learned software piece is integrated into the main code — most of the time manually — and never touched again, so that the systems are still unable to adapt to changing environment situations and don't show the desired flexibility.

In the light of such observations Mitchell (2006), one of the leading machine learning researchers, states his vision of machine learning by proposing the following longer term research question:

> Can we design programming languages containing machine learning primitives? Can a new generation of computer programming languages directly support writing programs that learn? In many current machine learning applications, standard machine learning algorithms are integrated with hand-coded software into a final application program. Why not design a new computer programming language that supports writing programs in which some subroutines are hand-coded while others are specified as "to be learned." Such a programming language could allow the programmer to declare the inputs and outputs of each "to be learned" subroutine, then select a learning algorithm from the primitives provided by the programming language. Interesting new research issues arise here, such as designing programming language constructs for declaring what training experience should be given to each "to be learned" subroutine, when, and with what safeguards against arbitrary changes to program behavior.
>
> Mitchell (2006)

For autonomous robots or other physically embedded systems the issue of integrating learning into a programming language is even more important than for other systems using machine learning, because the experience cannot be provided by the programmer, but must be acquired by the robot itself. For example when a robot is to learn how to navigate, no programmer can tell it which commands to use, because if this information was known, there would be no need to learn.

In this work we present a robot control language that explicitly supports learning in the program. It comprises all functionality connected with learning: gathering the experience, abstracting it, performing the actual learning process, and integrating the learning result into the program. Our Robot Learning Language (RoLL) supports arbitrary experience-based learning algorithms and can be extended easily with new developments in machine learning.

1.2 Scenario

In order to get a better intuition of how learning should be integrated into control programs, let's regard a typical scenario a household robot such as the one depicted in Figure 1.1 is confronted with daily. One task is to pick up an object, e.g. a cup, which might be implemented composed of atomic subtasks. The picking up task includes going to the object, gripping and lifting it.

1.2.1 The Challenge

For gripping an object, the robot has to navigate to a location that is easy and safe to reach and from where it can pick up the cup easily. Although the exact position of the cup is known, the robot has to decide where to navigate in order to grip it. First of all, there are physical restrictions as to where the robot can position itself. The cup is originally standing on some other object, possibly the worktop, so the robot must take care not to collide with the supporting object. But then there are still plenty of options. Should the robot stand directly in front of the cup? Then it might clash with the worktop. So it's probably better to move away a bit. But how much is "a bit"? Should the robot reach for the cup from a front position or from the side? This decision depends on the orientation of the cup, surrounding objects and which arm the robot intends to use.

The arm the robot intends to use? Another open question to decide. Not only that its answer relies on the position the robot chooses for starting the gripping process (we have a cyclic dependency here!), but we should also consider the later action of putting the cup down on the table. The case gets even worse when the arms aren't equally dexterous.

But not only decisions inside the picking up routine are crucial. There are situations where picking up an object is not possible, because it is out of reach for the robot. The higher-level plan should know about this. It should further know in which situations the robot fails to pick up the cup. Such a situation might arise when the state estimation is not accurate enough and the robot misses the object when gripping. A solution in the view of the top-level plan would be to add special sensing actions for determining the exact position of the cup and then start the picking up.

All these questions cannot be answered without considering the situation at hand. Studies in the cognitive neuroscience of motion control state that where skilled people grasp objects depends on aspects such as the clutteredness of the scene, the location of the objects, and the subsequent tasks to be performed. The grasp points for a bottle are cho-

Figure 1.1 Household robot working in everyday environments.

sen differently depending on whether the bottle stands solitarily or in the mid of a bunch of other bottles. The grasp points also vary depending on whether the bottle should be carried somewhere else or be used to fill a glass.

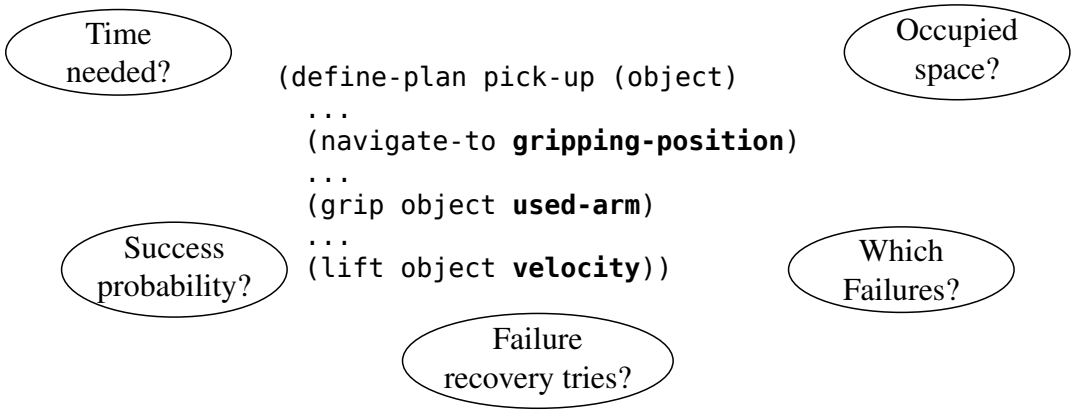
The question now is how to implement a robot program that adapts to the circumstances in the environment and performs sophisticated every-day activities reliably.

1.2.2 Solution Ideas

The first approach to make a robot pick up things is to regard the task as a control problem and program it by hand. However, we have shown that there are many open parameters apart from the control parameters steering the motors, which can hardly be determined analytically, as they involve many aspects of the current situation in the world. Besides, information about the routine like the time it will take to complete it or how often failures occur cannot be determined without observing the program execution. Researchers in autonomous robots agree that learning is the only way to make such routines flexible enough for human environments.

Instead of regarding the picking up process as composed of simpler actions, one could treat it as an atomic action, which is the view most planners have towards picking up. One could imagine a reinforcement learning approach, where the robot tries to grip objects and based on the information if it has succeeded adapts its control function. We have demonstrated some of the problems a robot has to deal with and which are critical for the success of the action. But when a reinforcement learning robot realizes that a try didn't work, it cannot attribute this failure to one of these aspects. Instead, it more or less randomly adapts its control function and tries again. The single problems of deciding on

Figure 1.2 Plan for picking up objects. Learnable parameter choices inside the plan definition are marked in bold font. Predictive models for this plan are shown as questions of the higher-level plans using pick-up.



a hand to use and where to stand are still there, they are just not represented explicitly. Thus, the learning is an unguided search in a huge state space. This needs a tremendous amount of trials and therefore time.

The critical questions in the pick up action can easily be identified by humans. So why not use this knowledge for speeding up the learning process? Figure 1.2 shows the plan for picking up objects and highlights the important parameters that have to be chosen. Besides, it illustrates the models a higher-level plan might be interested in. The atomic actions `navigate-to`, `grip`, and `lift` are also interesting parts to be enhanced by learning.

So instead of regarding picking up as a monolithic action, we use the structure of the program and learn small, easily learnable parameter decisions and models, which demands for much less experience than the approach ignoring the structural information. Learning such simpler functions is usually done by controlling the robot with a dedicated program for acquiring learning data, which is then fed into a learning algorithms producing an executable function. Then the program is equipped with this resulting function, never being modified again. With this procedure the programmer is freed from tedious parameter tuning and the decisions are adapted to the specific environment. However, real-world environments are highly dynamic. For example, when a coffee machine is added to the kitchen, the gripping and therefore the picking up task gets more difficult because of additional space restrictions. Or the robot might be equipped with a new arm, or the lower-level actions like gripping have been changed. When learning is performed only once, changes in the environment or to the robot cannot be reacted to appropriately.

As a consequence, control routines should be equipped with learning capabilities, which continually reconsider and adapt particular control decisions and subtasks.

The only way to fulfill these demands is to integrate learning capabilities into the language the program is written with. This allows a smooth interaction of programming and learning to use learned functions in a program where the structure of the activity is predefined. When learning is part of the program it is not performed as an external component, but an integral part of the execution. This allows to repeat the learning process at any time. This is why we developed RoLL, an extension of a robot control language with constructs for declaratively defining and automatically executing learning problems.

1.2.3 The Vision

We assume a learning robot to be equipped with a control program containing functions to be learned. At first, these decision functions are provided by simple heuristics, which

work in some situations and fail in others. Our robot constantly monitors the execution. The intention of picking up an object is the start signal for an interesting episode, which ends as soon as the robot has lifted the object to its desired position.

An *episode* is the behavior of the robot and the change in the environment over a specified time interval. The time interval is identified with by active processes of the robot control program and conditions in the environment.

During an episode, the robot can make an experience. Figure 1.3 illustrates that an experience is not just a data stream. It comprises the robot's intentions, beliefs, active processes, failures and internal parameterizations. All this is what we understand by the execution context.

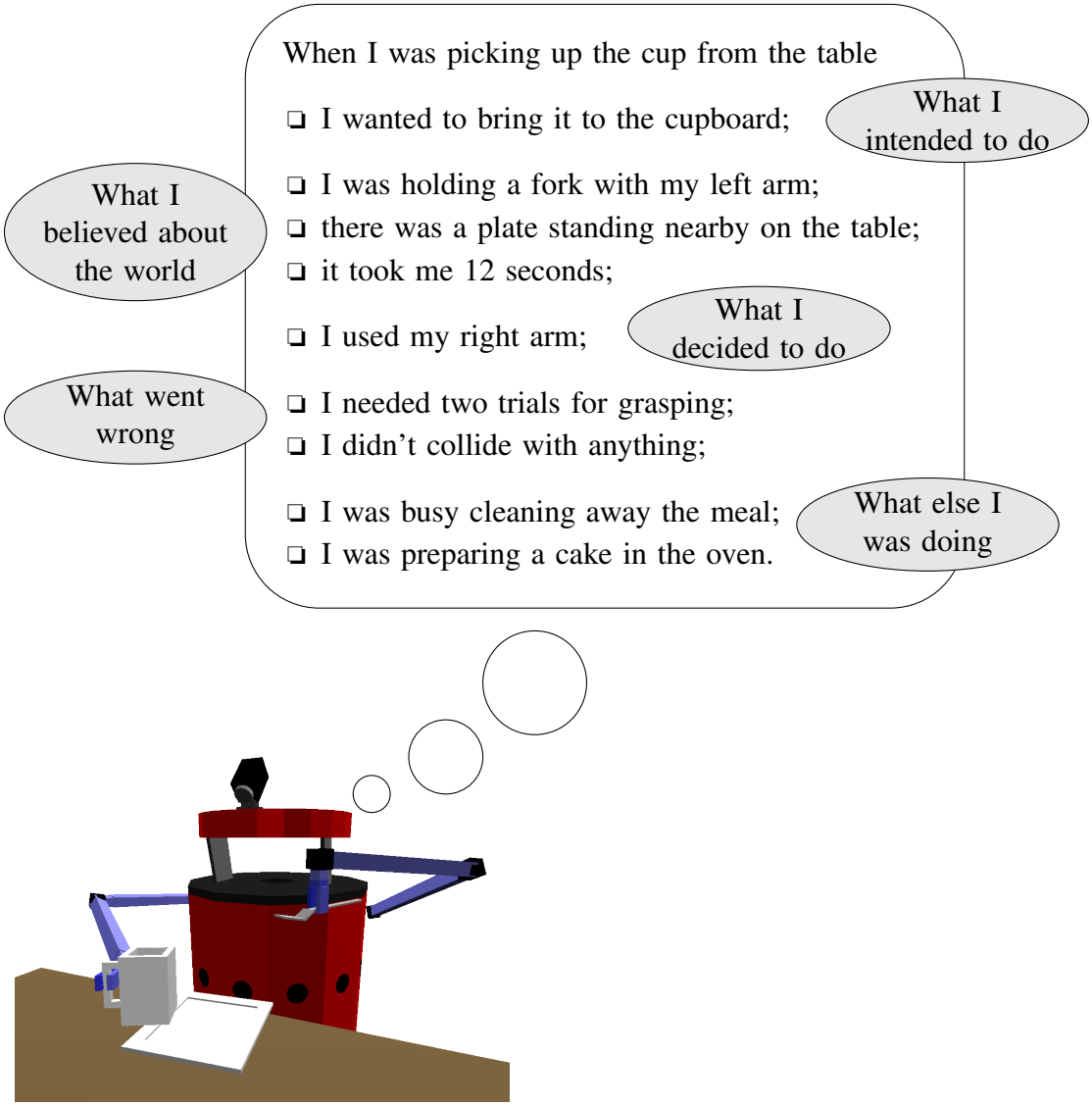
During the episode, the robot records several aspects of its activity performed in the world. At the start of the episode it notes the original object position and object type, the robot's position, the arm used for the picking up task, and the time stamp. All the time while the episode is valid, the arm position is recorded in each control loop cycle. Besides, all occurrences of failures are memorized with the associated time and failure type. Finally, when the end of the episode is detected, the time stamp is stored.

An *experience* is a compact, (partly) symbolic summary of a problem-solving episode for the purpose of learning. It may contain information about the intentions, the beliefs, and the perceptions of the robot. The experience also can contain information about events that happened in the episode and performance aspects such as resources needed or failures that occurred. In other words an experience should be comprehensive enough to generate explanations for the generated behavior.

As we work with a realistic robot, actions of observed episodes might fail. It may happen that the robot fails to grip the object or it slips from the robot's gripper while lifting it. The end of the episode is then detected, but the observed data might not be useful for learning all the problems described in the following. Therefore, such an episode should at least be marked as being interrupted. Other undesired effects during experience acquisition can be caused by low battery voltage or inaccurate state estimation.

At night, when the day's work is done, the robot uses its time to evaluate the experiences he has made during the day by learning from them. From the data observed during the gripping episode the robot can learn a variety of things. First, it can find a better strategy to select a hand when an object is to be gripped. It might have some experiences of when the lifting didn't work and a failure occurred. And it has seen different gripping episodes and can compare the times that they took.

Figure 1.3 Illustration of an experience. It includes intentions, internal parameters and beliefs of the robot as well as program failures and other active processes. The episode can be described with constraints on the same kinds of values, in this case on an internal process.

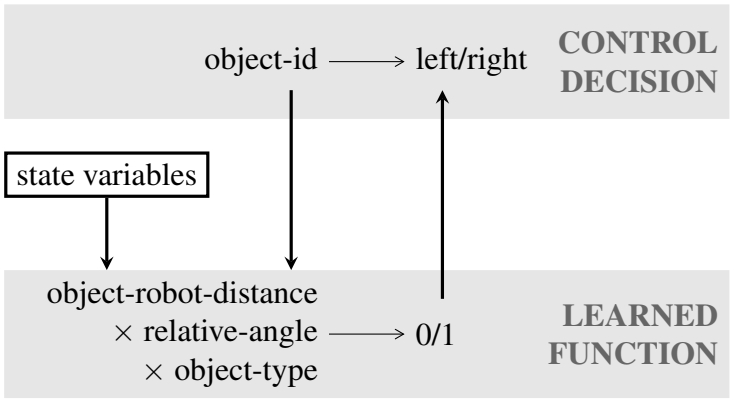


The *raw experiences* collected during robot execution are converted to learning task specific *abstract experiences* to make them usable for the learning algorithm and reduce the necessary number of experiences to be acquired. Raw experiences store all information about episodes that are useful for learning and can be abstracted in different ways to generate specific abstract experiences for different learning tasks.

To compare different gripping episodes, a good abstraction of the data would be to calculate the relative position of the object with respect to the robot. The duration of the task can be calculated from the start and end time stamps. By choosing only such gripping tasks that perform superior to other gripping episodes, a set of experiences — composed of the relative object position and the arm used — can be selected, which is then passed to a decision tree learner. The resulting function can select the best arm depending on the situation. When new experiences are available, the function can be updated with the new knowledge.

Other interesting functions to be learned from the observed experiences are models of the execution of the gripping routine. These include (1) the time it takes to grasp the object and lift it to the desired position, (2) the failures that might occur (possibly with hints how to avoid such situations), and (3) the space needed for moving the arm while gripping. All of these problems need different abstractions of the original raw experiences. Whereas the first problem would consider the relative position of the object toward the robot and the time the action takes, the second would probably take into account the absolute object and robot positions (because there might be objects in the way, which can only be captured

Figure 1.4 Embedding the learning result into the program. The learned function is represented in an abstracted form and has to be converted to the signature of the control decision.



with absolute position values) and the failure type. The third problem would evaluate the arm trajectory, i.e. the arm positions that where protocolled during the episode. It would consider an abstract description of this trajectory, for instance the bounding cuboid, as well as the situation description with the relative position of the object. For the second problem, a decision tree learner would be an appropriate approach, the other two learning tasks should rather be tackled with neural nets or extensions of classical decision trees like model trees.

In order to use the learned result, it must be embedded into the program. As the learning has been performed on abstracted data, the same abstraction must be calculated before calling the output of the learning system as illustrated in Figure 1.4. When the robot calls the function telling it which hand to use, the input from the control program's view are the robot's current position (given as a global state variable) and the object to grip. The learned function, however, expects a relative position of the object with respect to the robot and the type of object to be gripped. For calculating this mapping, the abstraction given in the learning process can be used. After calling the learned function, the result must also be converted to a form expected for the control decision. In the example, the left and right arm had to be mapped to 0 and 1 for the learning system to be learned, whereas the program works with the symbols `left` and `right`.

Embedding the learning result into the program means to make it available in the program and make sure that the calling parameters fit the ones expected by the learned function.

The next morning, the robot will perform better than the day before. When gripping objects it can now choose the hand to be used based on the experiences of the previous day. It can also forestall and avoid failures, predict if it might collide with other objects, and it has a notion of how long the action will take (which is useful for planning its activities and as an indication that an unknown failures has occurred).

Instead of making experiences during the normal execution, the robot could have used its free time to move around the kitchen and try certain actions. For instance, if it hasn't performed many gripping tasks during the day, it might carry objects around during the night to observe the experiences then.

1.2.4 The Bottom Line

We have pointed out the challenges of learning in the real world. The main problem in current systems is that learning is usually regarded as a way to improve a program from an external point of view. This approach cannot be accepted for continual, adaptable robot

learning. We think that learning parts of the control program must be initiated and executed by the program itself. A natural way to allow this is to extend a control language with constructs for learning. Embedding learning capabilities into a control language makes learning problems executable at run time and enables a smooth transition between programming and learning. Instead of solving a complicated, monolithic learning problem, several simpler ones can be solved and combined in the program. Also, before learning at all, simple heuristics for models or parameterizations can be programmed before they are replaced by more accurate learned ones.

1.3 Technical Challenges

The integration of a general learning procedure into a robot control language, which should be defined by declarative constructs poses several challenges both on the conceptual and on the implementational side. In particular they comprise

- the definition and automatic acquisition of experiences, as well as their abstraction and management,
- the integration of learning results into the normal control program, and
- to find a general mode of operation applicable to all experience-based learning mechanisms, so that the system works with arbitrary learning algorithms.

The first vital concept in learning are experiences. In the scenario we have illustrated that experiences are not mere chunks of data, but should contain the robot's intention and the execution context. This requires deep insight into the control program. In current systems, special data recording code is added to the primary program, which is clearly unacceptable for a programming language that should allow to add an arbitrary number of learning problems at run time. This demand poses two fundamental questions: (1) How can experiences be specified declaratively? and (2) How can the desired data be recorded without changing the rest of the program?

The specification of experiences must enable the programmer to define the desired execution context both from the robot's inner state and from its percepts, for example to specify an episode where a cupboard door is open and the robot is navigating. Having detected an episode, the robot must be told which data to record and at which intervals. All these instructions must be captured in a declarative specification to make it independent of the control program and to allow an intuitive specification of experience abstractions. Our solution is to use the concept of hybrid automata for modeling the control program and specify the experiences based on this model.

RoLL converts the experience specification into executable code. The code for observing and recording experience data runs in parallel to the main process and be able to observe all relevant aspects of the execution. This includes the robot's belief about the environment situation, its execution status (i.e. which goals are currently tried to be

achieved with which routines) and local variables in the routines. This is only possible with a programming language that admits insight into its control status and keeps track of its internal behavior. Such a language is RPL, which we use as a basis for RoLL.

The integration of learning results is not just a conversion from an external result format to a LISP function. Because the experience is abstracted before learning, the call of the resulting function must take into account these conversions. The data conversion needed for calling the function appropriately is very similar to the abstraction defined for the learning process, but some data that was observed during experience acquisition must be replaced by explicit input values of the result function. To avoid redundant specifications of abstractions, the integration must access the initial abstraction specification and combine it with a user statement as to which data is obtained in the same way as in the experience acquisition process and which is provided by the input.

With the demands on the definition of raw experiences and function integration, the concept of abstract experiences must on the one hand be similar to raw experiences and offer interfaces for experience abstraction. On the other hand it must allow to reconstruct the abstraction process in order to generate the embedded result function.

One important objective for RoLL was to support arbitrary experience-based learning algorithms. This doesn't only raise the question of how to enable a modular extension with new learning systems, but also makes it crucial to find a general schema of how learning problems are executed. The language should not be too specific in order to allow different learning systems. On the other hand, it should be possible to define all aspects of learning problems in the declarative constructs.

In sum, the challenges of developing a robot control language that includes learning constructs comprise the implementation of a modular experience acquisition and the design of a language that enables to define all aspects declaratively in a uniform framework.

1.4 Approach

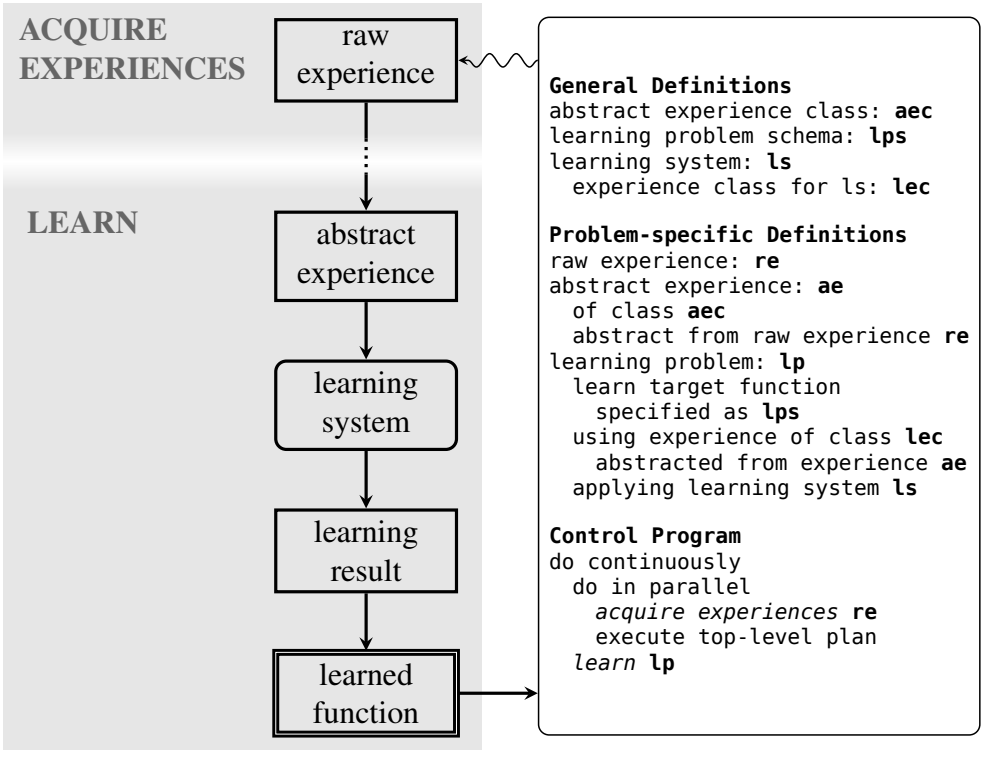
The language we present — RoLL (Robot Learning Language) — offers constructs for declaratively specifying all stages of the learning process: experience acquisition, experience preparation for learning, the call of the learning algorithm, and the inclusion of the learned function into the control program. Although we have motivated continual learning by a special class of learning problems (models and parameterization decisions), RoLL can be used for any kind of robot learning. As different problems suggest the use of different learning algorithms (e.g. decision trees for choosing the hand for gripping, neural networks for estimating the time needed to fulfill a navigation task), RoLL doesn't rely on a special learning algorithm, but supports the modular integration of any kind of experience-based learning paradigm.

Figure 1.5 gives a flavor of what RoLL looks like and how it works. The box on the

right contains the RoLL program, shown here in a simplified form. The left-hand side shows the learning steps when the program is executed. The program code consists of specifications needed for several learning problems (for example a learning system), the definition of a specific learning problem, and a control program. The control program executes some normal activity invoked by `execute top-level plan`. The execution of this plan is observed by the `acquire-experiences` command. When the daily activity has finished, the command `learn` uses the experiences with the specified learning algorithm, transforms the learning result to an executable function and integrates it into the control program.

The code specifying the experience acquisition should not be interwoven with the primary program. For finding an appropriate specification we needed a formal framework for declaratively describing the experience that is to be acquired. For this purpose, we use the concept of hybrid automata for partly modeling the control program and aspects of the environment and on the basis of these models describe the data wanted as an experience. Hybrid automata have shown to be a very effective concept for modeling continuous as well as discrete behavior of programs when executed in an environment. This allows the

Figure 1.5 Usage of RoLL. The program on the right side invokes the learning steps on the left side by calling the RoLL commands `acquire-experiences` and `learn`.



specification of a rich variety of experiences in RoLL including instantaneous changes in the environment or the internal program state as well as continuous observations.

1.5 Contributions

This work develops a novel robot control language with integrated learning capabilities. The main contributions are

- ❏ establishing a theoretical foundation of experiences using hybrid automata, and
- ❏ designing and implementing an extension of a robot control language resulting in a smooth interaction between programming and learning.

Both contributions are evaluated by applying the resulting language RoLL to different robot learning problems.

Many researchers don't regard the first contribution of analyzing what experiences are and how they can be represented explicitly as a pressing problem. Robot learning has been used in several applications and the architecture of learning agents is described in text books (cf. Figure 2.1). This knowledge is enough when single learning problems have to be solved, when experiences are obtained by writing special code that records the desired data from inside the program and this data is finally fed into a learning algorithm. When this process is to be automated, all parts of the learning process must be made explicit and be described declaratively. Therefore, finding a general description of experiences is a vital prerequisite for developing control languages that include learning capabilities.

The most difficult part is to describe experiences without having to modify the control program. To do this we model the execution of the control program using hybrid automata and define the required experience data according to this abstract description. We use hybrid automata further to describe the abstraction of experiences. The learned behavior models can be represented by hybrid automata, too, so that the complete learning process can be viewed in this framework. In this work, we use hybrid automata only as a tool for modeling aspects of the control program and defining components of learning problems. We do not make use of automatic verification techniques or stability proofs — topics which are amply discussed in the literature on hybrid automata.

The second contribution of this work is the specification and implementation of the language RoLL. It allows to define, execute and repeat complete learning processes. In particular, it enables a modular, declarative definition of experiences, which enables the automatic acquisition of experiences by observing the primary program. Another difficult subject is the integration of the learning result into the program. Not only compatibility between different systems must be ensured, there is also the issue of calling the learned function with the appropriate input, which must be converted similarly to the abstraction defined for the learning problem. RoLL is very flexible with respect to different learning systems, methods for storing and managing experiences, and learning problem types.

We used the implemented language on an autonomous household robot for learning while it was performing its activities. We show that RoLL enables the acquisition of experiences during the primary activity, its abstraction, the learning process and the embedding of the learned function into the control program. We demonstrate that the development of autonomous robots can be accelerated and enhanced by learning on the job and that our language constructs allow a declarative, general and easy to understand specification of learning problems.

1.6 Reader's Guide

This work introduces the language RoLL by first explaining its formal and implementational foundations, presenting the language and evaluating it.

Chapter 2 gives an informal overview of RoLL and the basic concepts that are needed throughout the work. This chapter is intended as a reference for the overall approach.

Chapter 3 presents the starting point for this work, which is the language underlying the learning concepts of RoLL. This underlying language consists of two layers: RPL, a robot control language developed by McDermott (1993) and the implementation of a belief-desire-intention architecture on top of RPL.

Chapter 4 explains the formal basis for RoLL by introducing the theory of hybrid automata and their application as a framework for robot learning.

Chapter 5 provides a detailed explanation of the language RoLL. We primarily show the concepts and their role in the learning process. A complete specification of the language is given in Appendix C.

Chapter 6 demonstrates the advantages of RoLL by providing a complete code example. Besides, we present some learning problems solved with RoLL and on the basis of them point out cases where RoLL is particularly useful.

Chapter 7 concludes by classifying continual robot learning in the context of a wider perspective of future AI and robotics approaches.

Appendix A is a short reference of the language RPL including only the concepts needed for this work.

Appendix B summarizes the definitions of hybrid automaton concepts.

Appendix C is a complete reference of RoLL. Together with the code example from Chapter 6 it can serve as a programming manual.

Appendix D shows how RoLL can be extended in several dimensions. It is intended as an example and tutorial for the RoLL extensions. At the same time it is a reference of the currently implemented extensions.

Some aspects of former versions of this work have been published in (Müller, Kirsch, and Beetz 2004), (Beetz, Kirsch, and Müller 2004), (Kirsch 2005), (Kirsch, Schweitzer, and Beetz 2005), (Kirsch and Beetz 2005), and (Kirsch and Beetz 2007).

Chapter 2

Integrated Robot Learning

This chapter gives an informal overview, first of how single learning problems are executed in RoLL, second how the learning is executed continually while the robot is performing its normal activities. Beside the system functionality we define the scope of this work.

2.1 System Overview

The central concept in our learning approach is the term "experience". In our framework, an experience is more than the data stored in log files, which is commonly used for learning. An abstract description of the experience concept was given in the scenario in the last chapter. In more technical terms we define an *episode* as all the available data observable during a stretch of time in the execution of the robot control program within the environment. An *experience* is a summary of the internal and external state changes observed during and associated with an episode. By summary we mean that not all available information is recorded, but only the one necessary for learning. State changes include events in the environment, the robot's control commands, its internal beliefs, decisions and intentions, as well as failures and the robot's reaction to them.

This notion of experiences enables operations on them, which are needed to reduce the number of experiences to be acquired and make the learning more efficient. One such operation is the abstraction of an experience into a form more suitable for learning. This can either be done in a linear fashion, where each raw experience is converted into one abstract one or in a network of experiences, where the raw observations are transformed to produce several abstract views. What is more, when the context of the experience is preserved, semantically sound methods for deciding which experiences are most useful can be applied to learn only with the most expressive experiences, thus enhancing the learning process as a whole (Kirsch, Schweitzer, and Beetz 2005).

To implement this notion of experiences in the learning process, our approach is based on two concepts: (1) the architecture of a learning agent as described by Russell and Norvig (2003) for a structured view on the learning process and (2) hybrid automata for a theoretically grounded, well-understood underpinning of experiences and their impact throughout the learning process.

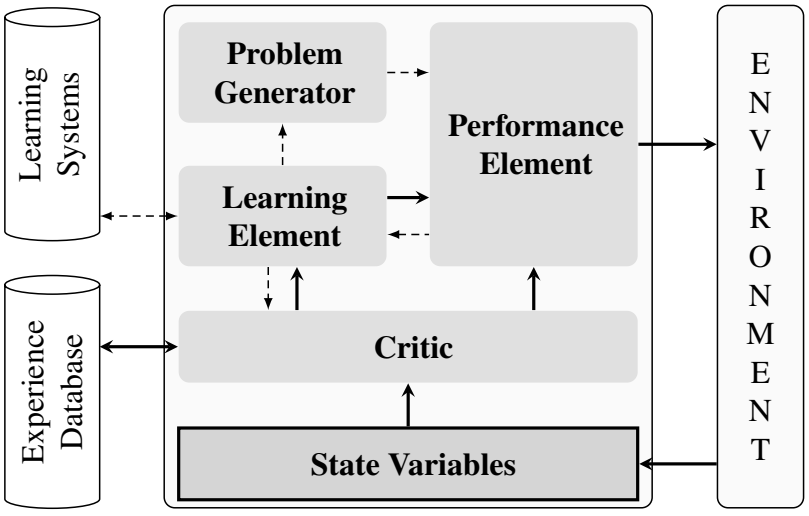
In the following, we first explain the agent architecture providing a structure of the learning process. Then we give a glimpse of what a program written in RoLL looks like and how the learning process is performed. We use hybrid automata for justifying and explaining the language constructs of RoLL, which would go into too much detail in this overview chapter. The use of hybrid automata as a theoretical underpinning for experiences is explained in chapters 4 and 5.

2.1.1 Architecture of a Learning Agent

The principal operation of RoLL is derived from the architecture of a learning agent described by (Russell and Norvig 2003) and depicted in Figure 2.1. It shows the basic interaction of the agent and the environment by exchanging percepts and control signals. The interpreted percepts are stored in global state variables. The agent architecture consists of several components, of which only the performance element is essential for controlling the robot. The other components are needed for learning or providing additional information to the controller.

The critic element can be regarded as an abstract sensor supplying the performance element with more abstract information than is contained in the state variables. This

Figure 2.1 Architecture of a learning agent after Russell and Norvig (2003).



includes the knowledge about the robot's own behavior in the form of models. In the context of learning the critic element's job is to detect episodes, record data and abstract, store, and manage experiences.

The learning element controls the learning process as such. It decides when and what to learn, what experience to use and which learning algorithm to employ. We assume that different learning algorithms are provided as external learning systems, so that new systems can be added and the learning process is not restricted to a special set of algorithms.

The problem generator instructs the performance element what tasks to fulfill in order to acquire useful experience. In this context, there are two questions: "Which experiences are needed?" and "How should the the robot choose its actions to achieve its primary goals and at the same time acquire useful experiences?" For the first question a tight interaction with the critic element is necessary for knowing which experiences are already present. Further, a deep insight into the impact of the robot's actions is needed in order to know beforehand that the proposed task will provide the desired experience. The second question is the well-known trade-off between exploration and exploitation. Intensive exploration will provide more experience for learning, whereas it might degrade the performance of the job at hand. A higher degree of exploitation usually gives better results of the primary task, but doesn't help to improve the performance in the future.

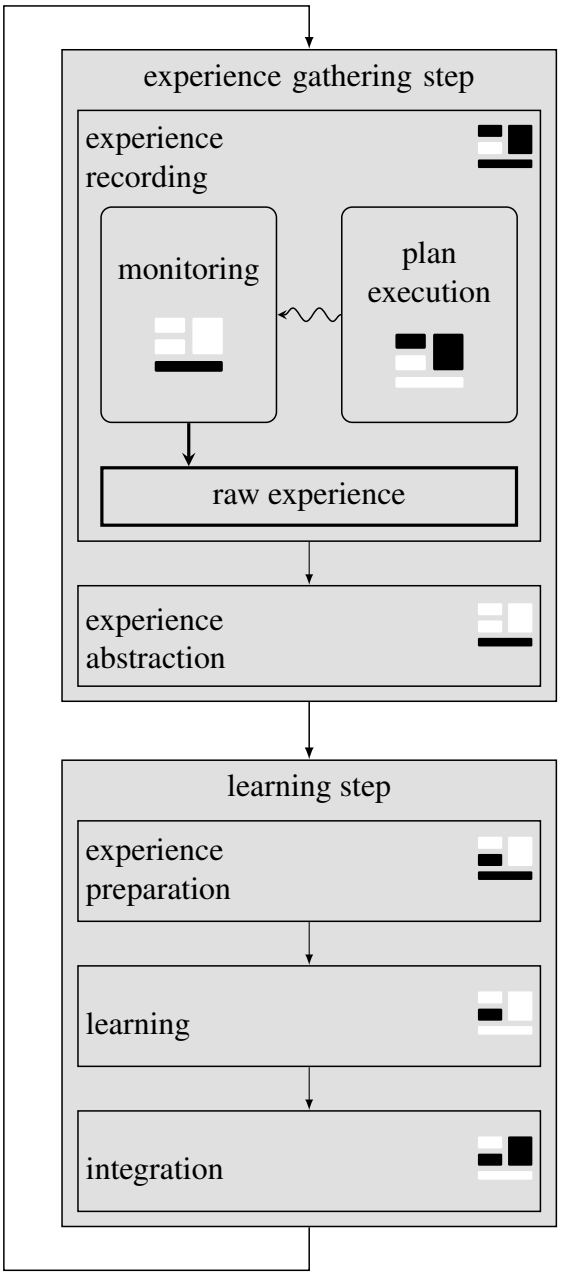
2.1.2 Learning Process

The architectural description of the learning process is mirrored in the way RoLL works. We now go into how RoLL solves single learning problems. In Figure 2.2 the left-hand side depicts the learning procedure in RoLL with references to the elements of the learning architecture. All operations in the context of experiences are handled by the critic element, the learning part is performed by the learning element, and the performance element is on the one hand used to produce observable episodes (together with the problem generator) and on the other hand receives the learning result. On the right side of Figure 2.2 the code pieces are shown that generate the respective behavior illustrated on the left.

A typical learning problem is specified and solved in two steps: first acquire the necessary experience, then use this experience to learn and enhance the control program. This process is usually repeated to make the learning result better and adapt the control program to changed environmental situations. The activity of these two steps is invoked by calling the commands `acquire-experiences` and `learn`, one possible control program could be the following:

```
do continuously
  do in parallel
    acquire-experiences re
    execute top-level plan
  learn lp
```

Figure 2.2 Learning process with reference to the elements of the learning architecture of Figure 2.1. On the right-hand side the corresponding code defining the learning process is shown.




```
raw experience: re
  observe execution of plan p
  recording variables
    (x,y) continuously
    (a,b,c) in case of success

abstract experience: ae
  of class aec
  abstract from raw experience re

learning problem: lp
  learn target function
    specified as lps
  using experience of class lec
    abstracted from experience ae
  applying learning system ls
```

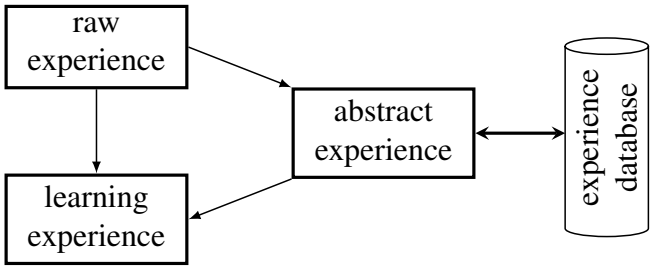
This program executes some top-level plan, e.g. doing household work. In parallel it observes the experiences defined declaratively as raw experiences. In this example, only one experience is observed, but RoLL supports the parallel acquisition of several experiences. Here we assume that the top-level plan ends at some point, let's say in the evening of each day. Then the experiences observed during the day are used for learning and improving the top-level plan before it is executed again.

The commands `acquire-experiences` and `learn` only tell the program *when* to observe and learn. The questions *what* to observe and *how* to learn are answered by declarative specifications, which are made independent of the control program and are depicted in Figure 2.2.

Let's have a closer look at the two steps involved in the learning process. The most intricate step of the experience acquisition is the observation of raw experiences during the execution of the control program. The observation process is independent of the robot's primary activity, but it needs access to all relevant information including local variable values and the execution state of the program. This process is modeled by hybrid automata and will be described in detail in the following chapters.

The freshly observed experiences must then be converted to a form that is suitable for the learning process. In principle, the raw experience can directly be transformed to an abstract experience, which can be passed to the learning system. But there are several practical reasons why the abstraction should take place in several stages, depending on the problem. One consideration is that experiences should be stored permanently so that the learning process can be repeated and build on former experience. Besides, for as experiences are valuable, they should be used for several learning problems. These requirements suggest to store the experiences permanently in an intermediate state that is abstract enough not to contain too many unnecessary details, but detailed enough to be usable for different learning problems. Therefore, the abstraction process usually looks as shown in Figure 2.3: The first conversion is performed directly after learning, its result

Figure 2.3 Typical experience abstraction steps. The raw experience is abstracted to an intermediate step, which is stored permanently. This experience can be retrieved from the experience database for different learning problems and be abstracted again for learning.



being stored in a database or some other storage device. When a problem is to be learned the experiences are retrieved from the permanent storage and adapted further to the learning problem and the learning system. The whole process can include an arbitrary number of abstraction stages.

The second step — the learning step — starts with the last operation on experiences by transforming them to a format accepted by to the learning system. Then the actual learning takes place by applying a specified learning algorithm to the experiences. The final step is to integrate the learning result into the program. This doesn't only involve the conversion of the learning result to a LISP function. Another question is how to use this function, because the learning has been performed on some abstraction, which is not necessarily the way in which the function is intended to be used. For example, when calling a navigation routine, the current position of the robot is known and the only input is a goal position. However, for learning one might have chosen an abstraction that involves the distance of the current and goal points. For calling the learned function, the input must be converted so that it fits the learning result.

After that the program runs with the modifications induced by integrating the learning result. The cycle starts again for more enhancements to the program.

2.2 General Robot Operation

We have just described an automated learning process specified in RoLL. We now give a wider picture of how single learning processes are integrated into the robot's overall mode of operation.

During the robot's activity its main contribution to learning is the acquisition of experiences. It observes all its doings on all levels of abstraction and records data that should be useful for learning later. It can for example record the outcome of actions in order to learn models.

But not only passive observation of experience is possible. Instead of only monitoring the robot's main process, it can further learn routines and plan parameters in an online way, so that its functions are updated each time the respective action is performed. This mode of operation is possible in RoLL, although we are more concerned with passive experience acquisition and offline learning in this work.

Besides, the robot constantly monitors its activity using the models it has already learned. On the one hand, this helps to detect execution failures, on the other hand, the robot can identify plans and routines that need to be improved. It is also possible that the robot detects its models' predictions not to be accurate enough, maybe because the environment has changed slightly. With this information, learning problems can be identified and scheduled for (re-)learning.

Equipped with new experience and a list of learning problems to be performed, the

robot's idle time is dedicated to learning. The first thing to do is to use the newly acquired experience for learning some of the problems scheduled for learning.

Some learning problems, however, require additional experiences, which cannot all be observed during normal execution. Especially when optimizing routines or plan parameters, more examples are needed. Therefore, the robot generates tasks allowing the observation of experience that is still missing for a problem at hand. These tasks can either be executed in simulation or in the real world according to the circumstances. The examples acquired in this way are then used for learning. Furthermore, the robot can use its idle time to perform online learning, e.g. reinforcement learning, on small, easily learnable problems.

Another method for not having to generate excessive amounts of experiences is their sophisticated management. By this we mean that experience should be stored permanently and in a less abstract form, so that it can be used for several learning problems, possibly for ones that have not even been identified yet. Experience handling also includes bookkeeping about the acquisition time of the experience. Thus, more recent experiences should be given more weight than older ones.

Overall, the robot's control program is composed of learned and programmed parts. During its normal activity experiences are acquired and necessary learning problems identified. These and previously recognized problems are learned either with the experience observed during the active time or with experience gathered by performing well-chosen tasks in the idle time. During operation there is no difference between learned and programmed parts of the program.

2.3 Scope of this Work

To our knowledge there is no other existing system that combines learning and programming in such a general way. This work can only be the beginning of a development in the direction of considering learning as an integral part of a robot's control program. We will now define the scope of this work in the light of the overall picture we have just given.

The structure of our control programs allows a seamless integration of learning and programming, because learning facilities are part of our robot control language. The language distinguishes plans, routines, functions providing plan parameters, and models as data structures. It makes no difference if any of these are programmed or learned, the usage is exactly the same. All of these types store certain additional information, which includes the origin and possibly status like *programmed*, *to be learned*, or *learned*. Thus, all of the basic types of subroutines can either be learned or programmed. Besides, formerly programmed parts can be replaced by learned ones without any changes to the rest of the program.

During program execution all relevant aspects of the execution can be observed in-

cluding percepts, internal local variables, low-level control commands, and the program's internal state, i.e. active processes and the calling structure of these processes. This monitoring takes place independently of the program that defines the robot behavior. Therefore, observing processes can be specified, added and removed without changing any of the activity generating part and it is possible that several monitoring processes are active at the same time. This modularity is an important requirement for making the whole program robust and easy to modify, even by the program itself. The monitoring facility can not only be used for acquiring experience for learning, but also for inspecting and evaluating the robot's activity.

In contrast to the overall approach described in the previous section we focus on learning during execution, mostly neglecting the generation of artificial experiences at idle times. However, we provide simple means for problem generation, where the programmer specifies in advance which parts of the state space should be explored. This is enough for imitating current learning approaches where the tasks to be performed are also preprogrammed. In the future this construct should be enhanced so that the robot itself can decide which actions provide useful learning experiences.

In our approach the learning problems are currently predefined and not generated automatically when needed. Therefore, the robot cannot decide to replace a programmed routine by a learned one when it realizes that the programmed routine performs suboptimally unless such a problem is already defined. Besides, the decision if a problem should be learned is either given beforehand or coded into the top-level program. So it is possible to acquire experiences, test the learning result and then decide if more learning is necessary. But this functionality must be programmed manually and is not part of the basic operating mode of RoLL.

As we have pointed out, experience is valuable when learning on the job. Therefore we store it in a database for later use. The experience gathered once can be added to later and it can be used for several different learning problems by defining different ways of abstraction. We also store a time stamp when the experience was acquired. In future work this management information will help to pick out the most informative experiences for learning.

The learning process itself is performed automatically according to the problem specification given by the user. Once a function has been learned it is integrated into the program and is ready to be used henceforth. From this point on it is treated as any other part of the program. If the necessity arises, it can be replaced by a newly learned function using the same learning problem specification or a modified one. At the moment, the programmer decides when the learning is to take place. In the future, the language could be enhanced to identify idle times suitable for performing the learning process.

RoLL is not restricted to special learning algorithms. We currently provide interfaces to a neural network learner and different kinds of decision trees. But the language offers constructs for easily including additional learning systems.

In sum, RoLL offers much of the functionality needed to tightly integrate learning capabilities into the robot control program. More sophisticated features like automatic detection of learning problems, target-oriented experience generation, and scheduling of learning in idle times are not supported explicitly, but can all be programmed manually within RoLL.

Chapter 3

Language Preliminaries

So far we have explained RoLL's functionality in an informal way. In doing so, we have implicitly assumed certain features of the original robot control language. In this chapter we describe these features in detail and show the available prerequisites for the implementation of our language extensions for learning.

3.1 RoLL Language Levels

In the last chapter we have presented the architecture of a learning agent (Figure 2.1 on page 17). Not all the activity of such an agent consists of learning. Therefore RoLL only comprises the architectural parts concerned with learning as depicted in Figure 3.1. The underlying control program must however fulfill certain requirements to be extensible with learning capabilities.

3.1.1 Prerequisites on the Performance Element

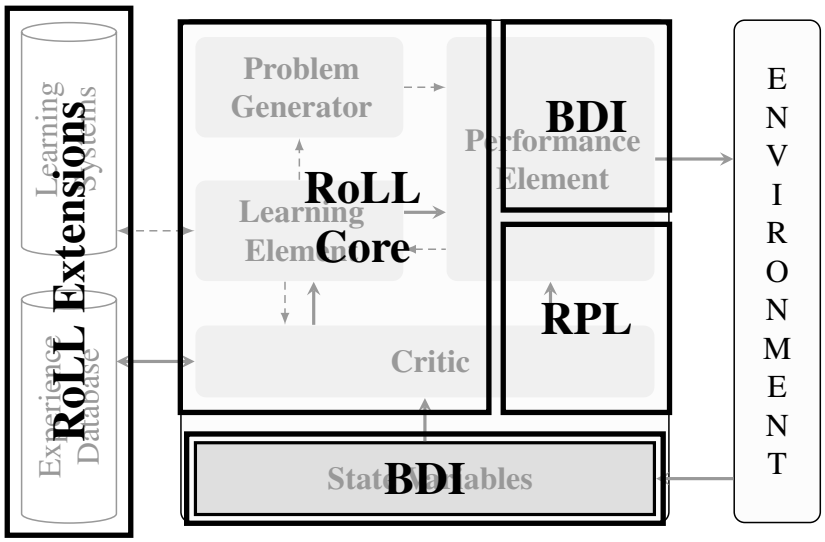
For non-learning operation the performance element is sufficient (possibly supported by parts of the critic element). This is also the part of the agent where things are modified when learning. Thus, the first prerequisite of the performance element is that the parts of the control program that are to be learned can be modified during execution. This means that it must be possible to overwrite existing procedure definitions and add new ones at execution time.

A secondary claim is to have additional information on procedures like if they have been learned, programmed or are still to be learned. We provide these requirements by representing plans, routines, and functions as data structures in an object-oriented framework. Their execution part is either a method or a function stored inside the data structure.

One of our main learning goals is to acquire models of the robot's environment and its own actions. These models should not only be used for prediction but mainly to improve the robot's performance. Therefore a program structure that makes ample use of models is an appropriate combination for the learning facilities provided by RoLL (although not a stringent prerequisite).

An ideal framework fulfilling these demands is the belief desire intention (BDI) architecture. According to the agent's belief it decides *what* it wants to achieve. By using models it then decides *how* to reach its desires.

Figure 3.1 Classification of RoLL language levels based on the architecture for learning agents.



3.1.2 Requirements for Experience Acquisition

Figure 2.2 on page 19 makes it obvious that the experience acquisition is a central part of the learning process. We have described the kinds of data we want to acquire: the current external state and local variables from the robot's internal state. While the first can be achieved easily by making the state variables global, the second is more tricky, especially with the claim that we don't want any special data acquisition code in our control program. A parallel process must be able to observe everything that is going on in the observed process, for example which hand the robot uses for gripping or how many people it expects for dinner (according to its knowledge and its models). Similarly, it must be possible to detect the start and end of processes inside the control program.

For example, we might be interested in the robot's navigation behavior. The monitoring process must be notified each time the robot's navigation routine is used.

In short, we need a sophisticated process management allowing parallel execution with a notification mechanism when certain program parts are activated and introspection methods for accessing all local variables within the program. This functionality is provided by the Reactive Plan Language (RPL) by Drew McDermott, which we use as the basis of our implementation.

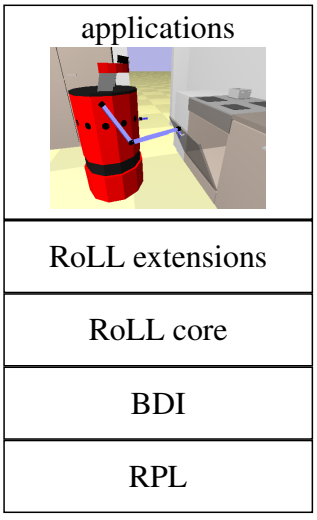
3.1.3 RoLL Language Hierarchy

The complete RoLL system consists of several language layers as shown in Figure 3.2. RPL constitutes the basic language layer (which is itself implemented as an extension to LISP). On top of this, we implemented the BDI layer to provide the necessary features of the performance element. Relying on the BDI constructs, the RoLL core language is the one that is the main subject of this work. These are the elements of the problem- and user-independent language. The functional aspects of the different layers are shown in Figure 3.1.

RoLL can and needs to be extended along several dimensions in order to be applicable to a wide variety of problems. One such dimension are the learning algorithms (called learning systems in RoLL, see Figure 3.1).

Another RoLL extension consists of the storing mechanisms of experiences. We use a relational database, because the data is easily accessible, manageable and can be cleaned by data mining before it is used for learning. But this is not mandatory. It is also possible

Figure 3.2 Hierarchy of RoLL language levels.



to store the experiences in log files or not to store them at all and use them for learning directly.

A third dimension of extendability of RoLL is not depicted in the figure. We have already mentioned some classes of learning problems: low-level routines, parameter settings, models, and functions selecting the appropriate routine. But RoLL is not restricted to those. In future work, it might be possible to tailor plan transformations to the learning paradigm (in fact, plan transformation is nothing else than learning, but the techniques used are still very different). Then one simply has to define a new learning problem class and tell the system how to integrate a learned plan into the program.

Figure 3.1 is a summary of the functional components in the light of the architecture of a learning agent. RPL provides the means for monitoring the program and the environment and serves as the underlying language of the BDI layer, which is used for implementing the programmed part of the control program. The core functionality of RoLL comprises experience acquisition and abstraction, the learning process and the integration of the learning result into the program. RoLL can be extended along several dimensions including experience storage systems such as databases and learning systems.

In the following we first introduce RPL and detail its features relevant for RoLL. Then we describe our intermediate-level BDI language that provides the basic architectural framework for the robot's normal activity, allowing the modification of the control program during execution. Finally we will give a summary of all the language levels RoLL is composed of.

3.2 Reactive Plan Language

The basic layer of the RoLL language consists of the reactive plan language RPL. We first introduce the concepts of reactive planning in general, before presenting the RPL concepts necessary for implementing RoLL. A summary of the RPL commands used in this work can be found in Appendix A.

3.2.1 Reactive Planning

When research on planning started, the execution of plans was thought to be much easier than the generation of plans (Gat 1997). However, it soon became clear that plan execution in real-world systems, especially on autonomous robots is a huge challenge. Actions can fail or produce unexpected outcomes and there might be unanticipated events during the execution.

Because of these problems, new models of plan execution were developed (Firby 1987; Agre and Chapman 1987; Payton 1986) under the name of "reactive planning". These

systems monitor the environment and check for possible execution flaws. The program reacts to unexpected events by changing the course of the lower-level commands.

These systems are employed for the execution layer of three-layer architectures (Gat 1997; Bonasso and Kortenkamp 1995), where the bottommost layer consists of the robot's basic skills in the form of a library. The top layer is the planner, producing plans that are made of very abstract actions like *go to the table* that aren't related to the actions implemented as low-level skills. Therefore an intermediate-level layer is necessary for executing the abstract plans on the robot, which is usually implemented as a reactive planning (or rather reactive execution) system.

The success of systems using this architecture can be mostly attributed to the execution systems, because the planning layer is often largely neglected. This shows that the execution is indeed a crucial part of autonomous robot control.

One of the best-known reactive planning systems is RAPs (Reactive Action Packages) developed by Firby (1989). Although RAPs are often used as execution layer in three-layer architectures, impressive results were achieved using RAPs without a planning layer on top of it. The Robot Chip (Firby, Prokipowicz, and Swain 1995; Firby et al. 1996) was able to pick up trash from the floor and remove it. Another very successful architecture is the Procedural Reasoning System (PRS) (Ingrand, Georgeff, and Rao 1990; Ingrand et al. 1996; Myers 1997). It has been applied to fault diagnosis and control of spacecraft (Georgeff and Ingrand 1989) and mobile robot control (Georgeff and Lansky 1987).

The Reactive Planning Language (RPL) (McDermott 1993; 1992), which forms the basis of RoLL, originally implemented the same ideas. However, McDermott (1990) denies 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)

Instead of using a planner working on an abstract level, the whole program consists of plans on different levels of abstractions. The plan generation of traditional planning is replaced by a transformational planner (Beetz and McDermott 1997). This approach subsumes the working of a traditional planner on the top level, but is more powerful as it works on all other levels of abstraction, too. All the plans in a program are written in a way that they can be reasoned about and modified.

To make this possible RPL keeps track of its execution status at all times. This means the program itself knows what plan it is executing, what goal the plan is trying to fulfill, and what data is used in the plan execution. This is the feature that makes RPL an ideal starting point for developing RoLL, because an experience consists both of data and the execution context in which it was observed.

RPL is implemented as a set of LISP macros, which means that the whole functionality of LISP is included. It also provides the possibility to define macros, so that it can be extended easily — a good starting point for implementing RoLL.

3.2.2 Language Overview

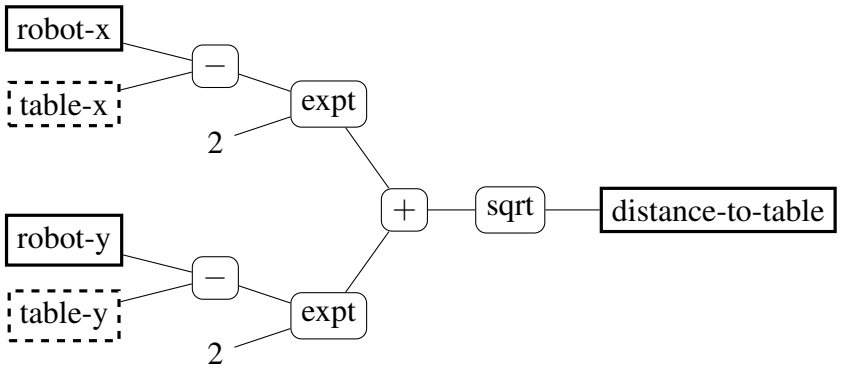
In the following we give a brief introduction to RPL, mainly addressing issues that are relevant for the language levels added for learning.

Besides the data structures of LISP, RPL has a special kind of variables called *fluents*. Fluents are variables whose values change over time. This is not unusual, but for fluents a change of value can be detected by other parts of the program. One application of this mechanism is to build networks of fluents. For example, when `x` and `y` are fluents, where the value of `y` is defined to be twice the value of `x`, `y` is adapted automatically whenever `x` changes, so that the condition of having twice its value is maintained. Fluent networks can be composed of arbitrary LISP functions with the construct `fluent-net`[1]. An example of a fluent network representing the robot's distance to the table is shown in Figure 3.3. The declaration creating this net is

```
(fluent-net `(sqrt (+ (expt (- ,robot-x table-x)  2)
                      (expt (- ,robot-y table-y)  2)))).
```

The whole expression is quoted, except for the variable containing the input fluents to the whole net. So when the robot changes its position, the new fluent is updated automatically. But if someone moves the table, this change will not be noted by the fluent net at once,

Figure 3.3 Fluent network calculating the robot's distance to the table. The behavior is determined depending on whether the table coordinates are given as fluents and if they are evaluated at recalculation.



[1] `fluent-net` is not a construct of the original RPL, but a later amendment for the development of RoLL.

because the table is represented by a normal, non-fluent variable value. Only when the robot changes its position again, the new table position is taken into account. In contrast

```
(fluent-net `(sqrt (+ (expt (- ,robot-x ,table-x)  2)
                      (expt (- ,robot-y ,table-y)  2))))
```

produces a fluent, where the table is assumed to stay where it is. If it is moved, this change will never be detected, because the table coordinates are evaluated at the time the fluent is created. Of course, the problems of a moved table can be avoided if the table coordinates are represented as fluents, too. In this case the second definition produces a fluent that is updated every time the robot or the table changes its position. On the other hand, fluents have more computational overhead than other values so that for immobile objects a non-fluent representation is the more efficient choice.

In real-world problems, failures are unavoidable and should therefore be detected and corrected. RPL represents failures as LISP classes, so that specialized failure classes can be defined. The construct `(fail :class failure-class)` represents a plan[2] that fails with the specified failure class. Failures can be handled implicitly by special RPL control structures or explicitly by the `with-failure-handling` construct[3] consisting of three parts: the *perform* part is the normal plan; the *monitor* part observes the perform part and generates a failure event if necessary; and the *recover* part, where reactions to failures can be defined. Such reactions include another try of the same plan or trying to achieve the goal with a different plan. Failures can also be handled implicitly by the control structures that we introduce in the following.

Unlike classical plan languages, where the plan is a partial order of actions, RPL provides a rich variety of control structures. The most straightforward one is `(if c a b)`, a plan that executes subplan `a` when condition `c` holds and `b` otherwise. Furthermore, plans can be executed sequentially or in parallel. There are several modes of parallel execution, depending on the overall reaction if one subtask fails. So `(par a b)` only succeeds if both `a` and `b` succeed, whereas `(pursue a b)` succeeds as soon as one of the subtasks has finished successfully.

There are special control structures for monitoring tasks. One such control structure is `(with-policy policy body)`, which executes `body` as long as `policy` is inactive. When `policy` is running, `body` is blocked. This construct is very useful, when a certain situation must hold while the robot is executing its main plan. For example, when the robot is to bring a cup to the table. Its main plan is the navigation towards the table. The policy watches over the cup in its hand. As soon as the cup is lost, the policy tries to recover it, while the navigation is stopped. Only after successful retrieval of the cup the main activity can proceed. If the policy or the body fails, the whole plan fails.

[2] Remember that every program (part) in RPL is a plan (see the beginning of this section).

[3] `with-failure-handling` is another extension, not included in the original RPL.

Moreover, RPL allows different kinds of loops (e.g. `loop`, `n-times`) and constructs that never terminate, such as `whenever`. `(whenever fluent action)` executes `action` each time the value of `fluent` changes from false to true. The `whenever` construct can be applied very effectively in combination with `with-policy` to monitor a plan.

Similarly to `whenever`, `(wait-for fluent)` waits until `fluent` becomes true. But this plan step only reacts once to the change in the fluent. For waiting for a certain time instead of a fluent value, RPL provides the construct `wait-time`.

3.2.3 The RPL Task Network

RPL is not only a sophisticated representation for plans. Since it was designed for plan transformations, it provides explicit access to its internal control status from within the program, so that a planner can understand the program and modify it during its activity.

This feature of understanding the program structure is mainly due to the explicit task network, which corresponds to the stack in normal programs, but is accessible from within the program and including tasks that have not yet started their activity. The plan `(seq a b)` is represented as a task with two subtasks. By this representation we have access to information about the plan itself, like its execution status (Has it started yet? Was it finished successfully?). Besides, we can address its subtasks and get information about them, too. In the following we will describe the information that can be obtained from the internal task description of a plan and then show how to navigate through the task tree in order to inspect other tasks.

First of all, a task has a status like *active*, *done*, *failed* or *evaporated*. This information can be useful when acquiring experience to find out if the observations are valuable (when the task has failed it is often better to discard the recorded data). Another useful feature of tasks is that they activate fluents when they start or end. Thus, the constructs `(begin-task tsk)` and `(end-task tsk)` return fluents that can be used as waiting conditions in `wait-for` or `whenever`.

More importantly, a task contains an explicit list of all of its local variables, called the environment. Let us consider the following plan:

```
(let ((hand-to-use (determine-best-hand object)))
 (grab-object object hand-to-use))
```

Assume that the variable `object` is known as an input variable from the plan containing this piece of code. The decision which hand to use for gripping is done by a function `determine-best-hand` that has access to the current values of the state variables. The variable `hand-to-use` is stored in the task's environment, so that it can be inspected by other processes and for example be recorded as training data. After the execution this information is lost and cannot be retrieved from the values of primitive percepts and commands, because it had emerged from the execution context.

A task also keeps track of its sub- and super-tasks. Therefore it is easy to navigate from one task to another one in the task network as shown in Figure 3.4. All subtasks of RPL control structures can be addressed by identifiers depending on the control structure. For example in the plan `(if c a b)` the task for plan `a` can be addressed by `(if-arm true)`. In a loop construct the iterations can be addressed by numbers, e.g. `(iter 3)` denotes the third iteration of a loop.

Another way to distinguish a unique task is to label it and later address it by this name. For example, in the task for `(seq a (:tag second-step b) c)` the second plan step can either be addressed by `(step 2)` or by its name `second-step`.

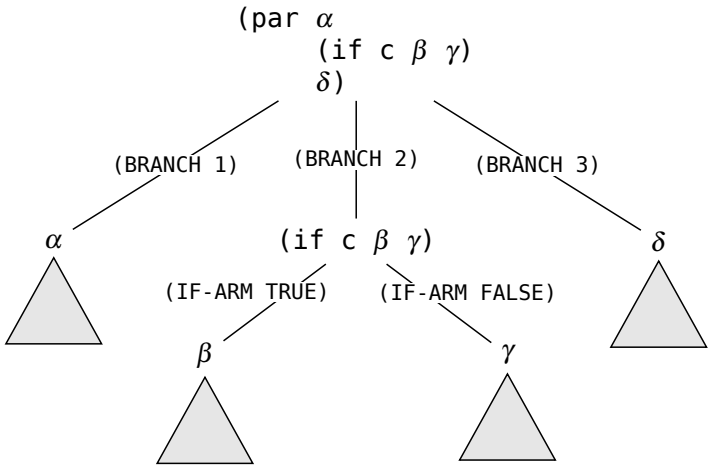
It is noteworthy that only individual, uniquely defined RPL plans can be addressed. In the plan `(loop a b c)` there is a group of processes `b`, one for each iteration of the loop. RPL doesn't arrange for addressing all of these iterations at once, which could be useful if we wanted to count how often the process is executed within the loop.[4]

3.2.4 RPL's Role in RoLL

RPL is an important prerequisite for the implementation of RoLL. The BDI level described in the next section is based upon RPL and extends its functionality for our purposes.

The explicit structure of RPL plans and the possibility to have access to the execution status from within the program are vital for the experience acquisition in RoLL. Because of the task network and the information kept therein, the experience acquisition can take place completely independent from the rest of the program, something which no other

Figure 3.4 RPL code tree of the plan `(par` α `(if c` β γ `)` δ`)` indicating the path names of the different subtasks.



[4] Nevertheless, RoLL provides this functionality, because it is indispensable for experience acquisition.

system can provide at present. With the concept of fluents, instantaneous reactions to changes in state variables are possible.

3.3 BDI Control Language

On top of RPL we built a language layer according to the belief-desire-intention model of agency. RPL was designed primarily for robot planning and plan execution, learning was not an issue. Because of the requirements pointed out in section 3.1 we had to extend RPL in a way that parts of the program can be learned and that the learned models are applied in a natural way. The language level described in this section can be used without the learning capabilities of RoLL, but as models play a central role, learning is very useful.

3.3.1 Belief-Desire-Intention Architectures

The belief-desire-intention (BDI) model of agency (Bratman, Israel, and Pollack 1991; Pollack and Horty 1999) is based on a proposition by Bratman (Bratman 1987) that an agent should have three mental attitudes: beliefs, desires and intentions.

Beliefs are the information that the agent has about its environment and its own internal state. In any realistic application the agent's belief is not complete, there is always information missing or imprecise. In some cases, this restriction comes from the agent's inability to sense certain information. For example, most robots lack the ability to smell so that our kitchen robot cannot smell that the cake has been in the oven too long. Another lack of information arises from inaccurate sensors. A good example is the information obtained by image processing. Depending on light conditions and reflections objects can be overlooked or hallucinated. Even if they are detected correctly, their position can be determined only with a certain limited accuracy. Finally, there are things the agent cannot know, because of physical restrictions or because the information is not yet available. An agent cannot know for sure how many people will attend dinner until they are all assembled, although it can make a good guess according to the family's habits and the information it has received.

The second concept of the BDI model are desires. A rational agent has to settle on some objectives it wants to achieve, although it might not be sure how to achieve them. Desires can be thought of the agent's motivation to do something. In the household domain typical desires would be not to break things, to fulfill the given tasks in a timely fashion, and to use resources economically.

The actual procedure of fulfilling desires is governed by intentions, the deliberative component of the agent program. Goals can often be achieved in different ways, so that the robot's intentions are a result of deliberation based on its desires and beliefs. For example, when the agent wants to have the table set at the right time, it reflects according

to its beliefs from where it can fetch the plates. Are they stored in the cupboard or are they in the dishwasher? Maybe they are still dirty and have to be washed before they can be used. In all of these situations the agent has to choose intentions according to the situation. However, the situation may change while the agent executes its intentions. While setting the table in the kitchen, it might receive the information that a guest is coming. What should it do? Revise its intention and set the table in the dining room instead, thereby risking not to have the table ready on time or keep to its intention to set the kitchen table, where there might not be enough room for all the people to eat comfortably.

Rao and Georgeff (Rao and Georgeff 1995) argue that all of these three mental states are necessary for agents working in real-time. The main problem BDI systems try to solve is when to reconsider intentions. One extreme would be to check the choice of current intentions in every cycle of the control program. This is of course very resource intensive and not applicable in real-time systems. The other end would be a strategy of blind commitment where intentions, once they are decided on, are executed to the end, no matter what happens during the execution. This option holds the danger of doing the wrong things, because the situation has changed so much that the original plan is not sensible any more. BDI systems try to find methods that only reconsider the intended course of action in situations where this reconsideration is reasonable (Schut and Wooldridge 2001).

BDI architectures have been used in several interesting projects over the last two decades (Georgeff et al. 1999). Applications range from software applications like computer game design (Li, Musilek, and Wyard-Scott 2004), workflow management (Pollack and Horty 1999), and air traffic control (Rao and Georgeff 1995) to hardware-based systems like space shuttle fault diagnosis (Georgeff and Ingrand 1989) and autonomous robot control (Georgeff and Lansky 1987).

The concept of BDI agents can be employed as part of a reactive planning architecture (see Section 3.2.1). Firby's (1989) RAPs already implement the idea that goals can be achieved by different routines that should be chosen according to the execution context. The Procedural Reasoning System (Georgeff and Lansky 1987) explicitly implements the concepts of BDI.

3.3.2 BDI in RoLL

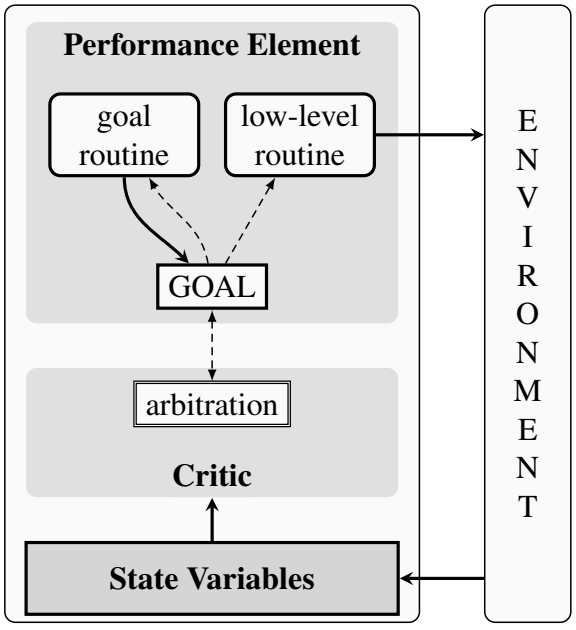
The focus of our work is different from the typical BDI application in that we are not so much interested when to reconsider intentions, but more how to choose them in the first place according to learned models. Besides, the BDI model makes agent concepts very explicit so that we represent them as objects instead of procedures and therefore make them easily modifiable by learning. Figure 3.5 shows the architecture of the BDI language layer based on the more general architecture in figure 2.1 on page 17.

We represent the beliefs by the global state variables and the part of the critic element that works during normal execution, that is, the part that manages and maintains models

about the environment and the agent's abilities. State variables represent the agent's percepts and can therefore be regarded as the low-level interface to the environment. Because the percepts are subject to constant change that should trigger agent behavior, we represent state variables by RPL fluents. For efficiency reasons it is also possible to declare a state variable to be non-fluent, which makes sense for values that are rarely changed, like the position of furniture. Unlike pure RPL fluents, state variable cannot only contain single values, but can be built up hierarchically as objects in order to give them more meaning and better possibilities for doing calculations with them. For example, we describe the agent's position by a class `2d-pose`, which contains the x- and y-coordinates and the orientation. Each of the slots and the whole object itself are fluents, so that both a change in the overall position as well as changes in the primitive values are noticed by other processes. If the robot turns, the fluents and processes depending on the x- and y-coordinate data are not updated, whereas the ones involving the orientation or the pose as a whole are notified by the change.

Goals define the agent's objectives in a declarative way. They include a description of the goal state itself (e.g. be at position (x, y)) and several parameters that are indispensable for the practical execution in continuous state spaces with nondeterministic, failure-prone actions like a timeout specifying the maximum duration until the goal is to be achieved and a success tolerance, so that for a navigation task not the exact point needs to be reached (which would be impossible), but all the positions in a certain range are accepted. The

Figure 3.5 Architecture of a BDI agent.



success tolerance depends on the context in which the goal is to be executed, for example it is crucial to reach a location accurately when the objective is to grasp an object. When, in contrast, the robot navigates somewhere in order to find an object, because it has seen it previously near this location, the accuracy is of minor importance.

For achieving goals, the language contains the concept of routines. There are two kinds of routines: goal routines, which can be thought of as plans posing new sub-goals, and low-level routines, which correspond to primitive actions that are executed directly. If a routine is implemented as a goal routine or a low-level routine depends on the design. Low-level routines are simple feedback loops calling one function over and over again. These routines can be learned directly with neural nets or reinforcement learning in the RoLL framework. However, they act as black boxes and their internal structure is not accessible. In contrast, goal routines have a finer structure with explicit subgoals, which allows more freedom in the achievement of those subgoals and a structure that can be modified by transformational planning. By the way, low-level routines can also be implemented outside of our language and be provided as an abstract command interface. This makes sense for very primitive actions like navigation and simple arm movements. Of course, these external routines cannot be replaced by learned ones later, since they are outside of the systems. It is however possible to learn models of them and thereby acquire knowledge about them. This enables well-informed choices between an external routine and one implemented within the language according to the goal and the current belief.

Routines are not only represented by executable code, but by objects containing additional information. Every routine specifies which goals it can achieve. This is not a guarantee that it will always achieve the specified kinds of goals, but it has the capabilities to fulfill them in certain circumstances. Besides, routines store information about their status with respect to learning, if they have been programmed, learned or are still to be learned.

Now we have described the components of the BDI language layer, but how does it work? First of all, an agent has certain top-level goals. These can be imposed by a user command or by a default plan that directs the robot's daily jobs. For achieving a goal, the routines that are able to reach it — at least in principle — are gathered. Among these routines the agent chooses one by calling an arbitration function. This function can either be learned directly and might contain a decision tree that selects the best routine in the current belief state. Or, the different routines are equipped with models about their performance with respect to the situation and based on these estimates, the most appropriate routine is selected. This selection is not only based on the belief state, but also on the goal. As in the navigation example above, sometimes a routine is better if it is fast, albeit not too accurate. When trying to reach other objectives like a subsequent grasping goal, it is more important that the routine be accurate, the time for reaching it being a secondary criterion.

In contrast to a full-fledged BDI system, we do not reconsider the choice of a routine

before it has stopped (either successfully or not). This corresponds to the strategy of blind commitment. Our focus lies on using explicit, learned models or arbitration functions and adapting them during execution. The expansion to a more intelligent reconsideration strategy is subject to future work.

Besides integrating the concepts of BDI agents into the language, this level of the RoLL framework includes some other useful features as an extension to RPL. The whole BDI concept is implemented in an object-oriented way. Although RPL uses the LISP object system for some features like the failure concept, it is not integrated for plans. In contrast, our routines are implemented as objects containing information on their learning status and the goals they can achieve. The execution component of routines is implemented by a one common RPL method `execute` invoking the routine behavior. We have also demonstrated how fluents can be combined with the object system by allowing fluents for all compositional levels of state variables.

Another convenient extension is the use of explicit units for numbers in the program. So a navigation goal is not posed by declaring "go to position (2,10) with orientation 2.13", but by saying "go to position (2m,10m) with orientation 122 degrees". This seems quite trivial, but really makes life easier, especially when dealing with different units, for example radian and degrees for angles. The Mars Climate Orbiter (Mishap Investigation Board 1999) has proven that misunderstandings about units can lead to disastrous failures.

In sum, the BDI language level is a language built upon RPL that includes the basic concepts of belief-desire-intention agents. It can be regarded as a stand-alone language, but can be used more efficiently in the combination with RoLL, as RoLL allows the learning of the models indispensable for the BDI concept to work. Our implementation establishes necessary prerequisites for integrating learned parts smoothly into the existing code.

3.4 Summary

We have explained the different language levels RoLL builds upon. Everything is based upon RPL. The BDI level contains useful extensions to RPL and implements the concepts of the belief-desire-intention model of agency. The RoLL core level contains the indispensable learning components concerning the experiences and the learning process as such. RoLL extensions are necessary to make the system work, but the question which ones are used or even included is subject to the user's preferences. On top of all this, different applications can use the RoLL system with all its extensions.

Having now presented the prerequisites of the underlying language levels, the next chapter addresses the theoretical background of the RoLL core language and Chapter 5 will describe the core language and extensions in detail.

Chapter 4

Hybrid Automata for Learning in Autonomous Systems

Chapter 2 has provided a general understanding of the steps involved in continual robot learning and the language elements of RoLL. One challenge of this work was the design of the language so that all these aspects can be expressed declaratively and comprehensively. In this chapter we introduce a formal framework — hybrid automata — providing the basis for the RoLL constructs. After motivating the need of such a formalism and its relation to RoLL, we introduce hybrid automata and variants of them. After that, we show how the language levels described in the last chapter can be modeled by hybrid automata as a basis for the experience acquisition of RoLL. Finally, we show how the whole learning process can be described in the framework of hybrid automata.

4.1 A Model for Robot Learning

The design of RoLL required declarative language constructs for all steps of the learning process: the identification of program parts that can be learned, the acquisition, abstraction, and management of experiences, the learning process itself, and the integration of the learned function into the control program. The specifications needed for learning should be comprehensible and universal so as to comprise all possibilities arising in arbitrary learning problems.

The biggest challenge lay in the experiences, how to describe their acquisition in terms of observations and how to abstract them for learning. Experiences are composed of external and internal observations. The first correspond to the agent's belief state, the latter comprises its execution status (i.e. which goals and routines are currently active) and its internal state (i.e. the local program variables). Another dimension lies in the timely distribution of data that is to be recorded. For some experiences, one-time observations

are necessary, like "How many people were in the room at the beginning of the meal?". In other cases, continuous observations are more appropriate, for example "How often does the robot lose the knife while it is cutting something?". Of course, combinations of those two cases are required as well.

The basic idea for describing the experience acquisition is to define an abstract model of the execution of the agent program within the environment including the internal state. On the basis of this model, episodes are identified and the relevant data for an experience is described.

4.1.1 Hybrid Systems

We have seen that it must be possible to model both discrete changes (the meal starts) and continuous processes (the cutting activity). A hybrid system is characterized exactly by these two aspects (Branicky 1995), so that the description of the outside happenings as well as the agent's internal state can be modeled in a natural way by the notion of hybrid systems. The external process contains discrete state jumps in the form of events like going through a door. Here the state of being in one room changes abruptly to the state of being in the next room. Inside the robot discrete changes correspond to procedure calls of any kind. Continuous effects are numerous in the environment, for example the process of boiling water. The water constantly changes its temperature until it reaches its maximum temperature, from where on it remains steady. Inside the program, the execution of a plan or routine is a continuous process.

The interaction of subsystems with continuous and ones with discrete behavior occurs in many contexts, for example chemical processes, traffic control, manufacturing, in short in all real-world domains that are manipulated by discrete control (Alu et al. 1999; Branicky, Borkar, and Mitter 1998; Henzinger and Wong-Toi 1996). Because hybrid systems are so common and the motivations for modeling them are diverse, there is a rich assortment of literature on the topic (Antsaklis 2000), many of which examine theoretical properties and automatic verification of hybrid systems (Alur, Henzinger, and Ho 1996; Alur, Dang, and Ivancic 2002; Branicky 1993).

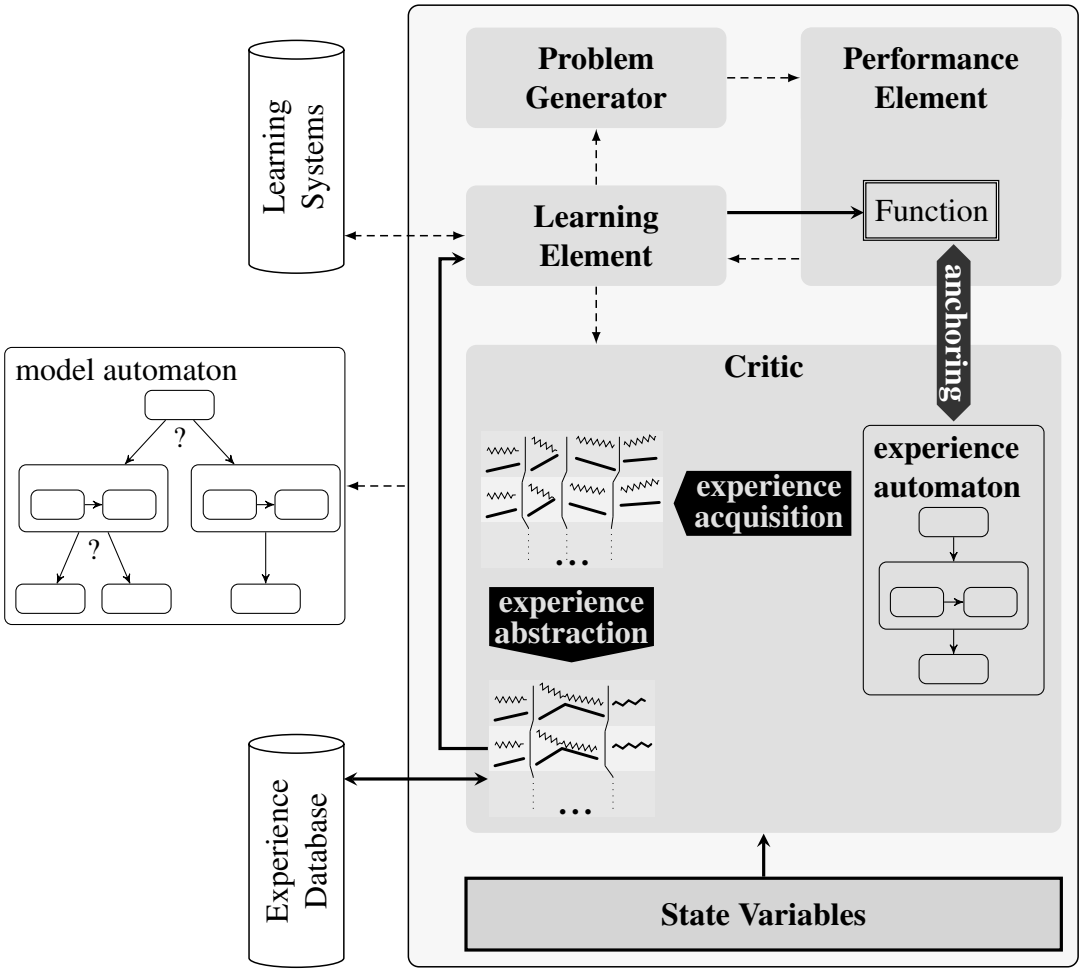
For developing RoLL we are neither interested in formal properties of hybrid systems in general nor in proofs about the behavior of a hybrid system. What we need is a well-understood, comprehensible framework for modeling a hybrid system. One way of specifying hybrid systems are hybrid automata (Alur et al. 2001; Henzinger 1996), which we chose as the underlying framework for RoLL. One reason is that automata are an ostensive tool for modeling hybrid systems. Besides, their definition can be extended in a straightforward way to encompass hierarchical and probabilistic concepts.

4.1.2 Hybrid Systems for Automated Robot Learning

Hybrid automata form the basis of the RoLL language design. The main motivation was the specification of raw experiences, but we show that the hybrid automaton concept can be carried over to other aspects of learning and even to the implementation of the robot control program.

Figure 4.1 illustrates the use of hybrid automata in the learning process on the basis of the learning architecture presented in Figure 2.1 on page 17. The first step in the learning procedure is the acquisition of raw experiences. For defining the experience, an experience automaton is defined and anchored to the control program executed in the environment. Based on this hybrid model, the data to be observed is defined. This data can stem from

Figure 4.1 Architecture of a learning agent when using hybrid automata.



external observations (the state variables) or from internal information about the program execution (active processes and local variables). Each detected run of the experience automaton is identified as an episode. Data associated with the episode can be recorded once at the beginning or end of the automaton execution or constantly during the interval the automaton is active. With a hierarchical nesting of hybrid automata this provides a rich description of the data observed. In Figure 4.1 different episodes of data are shown in different shades of gray. The vertical lines separate the data observed during the run of sub-automata and can therefore be thought of as sub-episodes. The thin zigzagging lines visualize external data, the thick straight lines the data gathered from the program execution status.

In the experience abstraction step the structure of the hybrid automata is maintained or adapted to a structure that is semantically sound in the abstracted experience. Not only the automaton structure is changed in the abstraction process, but also the data representing an automaton run. This transformation of automata gives a very expressive language for abstracting experiences. In the figure, the hierarchical structure of the abstract experience is changed and internally and externally observed values are combined (indicated by wider zigzags).

The abstraction level used for learning can require a specific structure of the automaton of the learning experience, for example when learning with a neural network the learning system may assume that the data representing the beginning of the automaton execution contains the input values of the net and the data observed at the end of the automaton run represents the output value.

In the current implementation of RoLL the learned function is integrated into the control program without any more reference to hybrid automata. However, the things we want to learn in RoLL are mostly models of the robot behavior. These models can best be represented in the light of hybrid systems. For closing this gap, one would only have to model the control program with a hybrid automaton skeleton, i.e. specify the structure, but omit quantitative details as shown in the model automaton in Figure 4.1. This automaton can then be replenished with learned prediction models to make accurate behavior predictions possible and allow a uniform access for using the models.

If we describe our program as a hybrid system, we soon realize that the system can be modeled in several ways. One way would be to describe the top-level program by a sequence of several continuous processes. The processes correspond to sub-plan invocations, which are typically extended over a period of time. Then we have one hybrid system, where the discrete changes occur when one sub-plan has finished and another one starts and continuous processes are captured as a black box in the sub-plans. However, we might want a deeper understanding of why a sub-plan produces the continuous behavior we observe. This can be done by having a look inside the sub-plan, which is built up in the same way as the top-level plan: it contains calls to sub-plans, which again show continuous behavior. This means that the continuous behavior of plans can either be specified by

a black box view or by opening the box and having a look at the hybrid system contained inside.

External processes or situations produced by the agent's behavior also exhibit a hierarchical structure. For example the task of setting the table can be interpreted by one continuous state that has a certain duration and whose result is that the table is set. On the other hand, we could have a closer look and describe the process of table setting as a sequence of continuous actions: getting the plates, carrying them to the table, putting them down, etc. The single actions in this hybrid systems can in their turn be expanded to get more insights.

Another noteworthy point is that the same processes can be described by different hybrid systems, depending on the information wanted. For example, we can regard the process of setting the table from an internal point of view and be interested in questions like "In which sub-plans do failures occur?" or "Which sub-plans are invoked how often?". For these questions the internal calling structure of the table setting plan must be known and observed. On the other hand we could discard all the internal information we have and put ourselves in the place of an external observer that doesn't know anything about the internal program structure. This description could give answers to questions like "How long does it take to set the table?" or "How many objects are moved?". In both cases, the robot performs the same actions, but the models of this activity are quite different.

In sum, we can understand the robot's program and its execution in the world as a hierarchical hybrid system, which we want to use as a basis for specifying the experience that should be acquired. Of course, for this specification it would be unmanageable to model the whole program as a completely expanded hierarchy of hybrid systems, and it isn't necessary after all. We allow to specify the interesting parts of the agent program and its execution as a hybrid system with subsystems to an arbitrary granularity. The hybrid automaton model then serves as a basis for describing the desired experiences.

4.2 Types of Hybrid Automata

Hybrid automata are a means of modeling hybrid systems, which are ones that contain both continuous and discrete change, for example autonomous robots. Originally hybrid automata were developed for system verification (Alur et al. 1993; Alur, Henzinger, and Ho 1996). The focus of these works is to guarantee certain features of the modeled system, whereas we only use them as a modeling tool. Therefore our definition of hybrid automata differs slightly from the classical approaches.

We have sketched the purposes for which we need hybrid automata: (1) modeling of the control program within the environment for experience acquisition; (2) automaton transformations on experiences; (3) modeling the agent program for identification and

integration of control program parts to be learned. Overall, hybrid automata constitute the formal underpinning of RoLL and are used as the basis for the RoLL syntax, especially in the context of experiences.

These applications require several extensions to the classical definition of hybrid automata. As argued in Section 4.1 our view of the system is hierarchical, which we have to model with the hybrid automata. The third purpose additionally requires a probabilistic notion of hybrid systems. In the following we describe different types of hybrid automata needed in this work. After that we give a brief summary of the automaton types and their application in RoLL. As a reference, all the definitions presented in this chapter are summarized in Appendix B.

4.2.1 Basic Hybrid Automata

A hybrid automaton (HA) is defined by a tuple $\mathscr{H} = \langle V,\ M,\ flow,\ T,\ act,\ cond \rangle$. V is a finite set of variables. The *data state* of $\mathscr{H}$ is a function $\sigma : V \to D$ assigning each variable $v_i \in V$ a value from some basic set D, e.g. the real numbers. The set of all data states is called Σ.

The interaction between continuous change and discrete jumps is represented by the set of modes M and the set of transitions $T \subseteq M \times M$ between modes. The modes capture continuous behavior, whereas transitions model discrete modifications of the data state. For example, when modeling a control program, a mode corresponds to the execution of a procedure and transitions model procedure calls. The *program state* of a hybrid automaton $\mathscr{H}$ is a function $\gamma : M \to \{0,1\}$ assigning a value 0 or 1 to each mode. Stated differently γ is the characteristic function of the set of active modes in $\mathscr{H}$. The set of all program states is Γ.

An *automaton state* θ of a HA $\mathscr{H}$ is a tuple of a program state and a data state over the variables V of the automaton: $\theta \in \Gamma \times \Sigma$. The set of all automaton states is called Θ.

The $flow$ function describes the continuous behavior of the system. It gives the changes of the data state depending on the time. As robots usually work with discrete time, which is determined by the percept update cycle, we can define $flow$ as a function assigning each mode a function that maps the old data state σ to a new one σ': $flow : M \to (\Sigma \to \Sigma)$. In the literature the $flow$ function is often restricted to linear differential equations. This makes the verification of hybrid systems feasible. In our case such restrictions are unnecessary, because we either draw samples from the continuous behavior as experiences or learn the $flow$ function for later predictions. In both applications we don't need a mathematically suitable representation like differential equations. In most cases the function can even be unknown, which makes it necessary that we allow the value UNKNOWN instead of a function: $flow : M \to \text{UNKNOWN}$.

The functions *act* and *cond* specify the effects and occurrence conditions of transitions. $act : T \times \Sigma \to \Sigma$ returns a new data state σ' when the transition occurs from the

current data state σ. The *cond* function assigns a predicate to each transition. When this predicate is true, the mode jump occurs, otherwise the program state remains unchanged. Normally, the definition of hybrid automata contains an additional function giving the invariant of a mode. Transitions occur when invariants are violated. As we aren't interested in assertions about mode properties, we model the invariants in the form of jump conditions. Therefore discrete changes solely depend on the *cond* function of the transitions.

A hybrid automaton works as follows. We start with a given automaton state, i.e. the variables have certain values and one mode is active. The current mode is connected to other modes by a set of transitions. In every time step (as we have pointed out we can assume discrete time) the variable values are adapted according to the $flow$ function. When a jump condition from the current mode to another mode is fulfilled, the program state shifts so that the previously active mode becomes inactive and the new mode is activated. If jump conditions to several modes are fulfilled simultaneously, all transitions are executed and the target modes are all activated. The data state is modified according to the *act* functions of the applicable transitions. Now we have another automaton state and the process starts anew, possibly in parallel if the automaton state contains several active modes.

We have now presented the common definition of hybrid automata. Unfortunately the behavior that can be represented by this notion is not enough for RoLL. In the following we extend this plain variant of hybrid automata in different ways.

4.2.2 Hierarchical Hybrid Automata

As motivated in Section 4.1 we need a hierarchical version of hybrid automata (HHA). The hierarchical structure of RPL plans cannot be described in a meaningful way without a hierarchical structure. Alur et al. (2001) present a hierarchical version of hybrid automata, which is constructed for automatic verification. For the hierarchy we need, we simply equate automata with modes. This has several implications for the definition.

The set M denotes the set of direct sub-automata of $\mathscr{H}$. The variables of an automaton are also variables of the subautomata: $\forall \mathscr{M} \in M : V_{\mathscr{H}} \subseteq V_{\mathscr{M}}$[1]. Because of the hierarchical activation of modes, M must contain a starting mode $\mathscr{S}_{\mathscr{H}}$ and a terminating mode $\mathscr{T}_{\mathscr{H}}$, unless M is empty, i.e. the automaton is only described by a $flow$ function. The starting mode is activated as soon as the parent automaton $\mathscr{H}$ becomes active. The execution of the parent mode ends when the terminating mode is activated.

As a consequence, we define the program state of a HHA $\gamma : M \cup \mathscr{H} \to \{0,1\}$ assigning a value 0 or 1 to each mode and the automaton itself, which is characteristic function

[1]The notation $V_{\mathscr{H}}$ denotes the set of variables of automaton $\mathscr{H}$. We use it only when a clarification is necessary as to which set of variables is meant. This notation is applied analogously for other parts of the automaton definition.

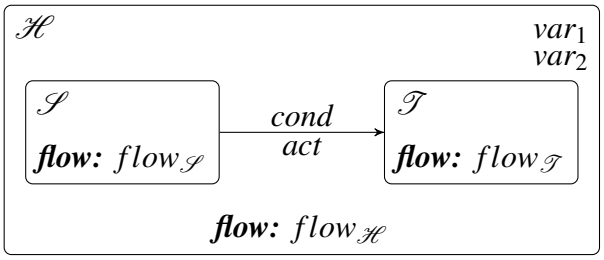
of the set of active automata in $\mathscr{H}$.

The $flow$ function is now defined as $flow : \Sigma \to \Sigma$ in every automaton. This means that the flow function is not associated to a mode in the definition of the parent automaton, but each automaton has exactly one flow function describing its behavior. The flow functions of the modes are defined in the automaton definitions of the modes, being automata in their turn. As a consequence, even the top-level automaton can be described by the changes it effectuates in the data state or by its structure, i.e. sub-automata and transitions between them. There is a unique connection between the automaton structure given as modes and transitions and its flow function. Normally, this interrelation is very complicated so that the flow function has the value UNKNOWN in the higher layers of the automaton. When the modeling becomes very detailed, it makes sense to replace further explicit modeling by the $flow$ function, which is then easy to approximate.

Besides the hierarchical composition of modes and the resulting reinterpretation of the $flow$ function, the definition of hybrid automata applies unchanged to the hierarchical version. For a complete definition of hierarchical hybrid automata (HHA) see Appendix B.

Figure 4.2 depicts a HHA with two subautomata. $\mathscr{S}$ starts at the same time as the whole automaton, whereas $\mathscr{T}$ is the terminating mode. The variables var_1 and var_2 of automaton $\mathscr{H}$ can also be modified by the $flow$ functions of $\mathscr{S}$ or $\mathscr{T}$ or the *act* function assigned to the transition between the two. The overall flow function $flow_{\mathscr{H}}$ is a composition of the flow functions $flow_{\mathscr{S}}$ and $flow_{\mathscr{T}}$ in combination with the behavior of the automaton, i.e. the functions *cond* specifying when the state transition takes place and *act*, which effectuates the state transition.

Figure 4.2 Hierarchical hybrid automaton.



4.2.3 Probabilistic Hybrid Automata

In order to model real systems appropriately, probabilistic state transitions are indispensable (Beetz and Grosskreutz 2005; Henry 2002). The probabilistic hybrid automaton (PHA) model assumes that the discrete mode jumps don't occur according to propositional rules as the *cond* function would require, but are governed by probability distributions on the possible successor modes. We therefore modify our original definition of

HA by replacing the function *cond* with the function $prob : T \times \Sigma \rightarrow [0,1]$ assigning to each transition a probability table, which means that a transition only occurs with the probability given in the probability table according to the current state. For a given jump condition σ from a given mode $\mathscr{M}$, the probabilities of different outcomes must sum up to 1: $\sum_{\mathscr{M}'} prob((\mathscr{M}, \mathscr{M}'), \sigma) = 1$.

With this extension, mode jumps occur depending on certain conditions in the data state, but only with a certain probability. This is more realistic for real-world applications. In the context of RoLL we use HHA mainly for experience acquisition. In this case, it is of no importance why a transition occurred, we are only watching the automaton. Therefore the probabilistic extension is not necessary here. It becomes crucial when we embed the learning language into a wider context, where we also model the functions to be learned as HHA. Here we can identify the learning problem of getting the transition probabilities in order to be able to predict the robot's behavior in the future.

4.2.4 Summary of Automaton Types

In this work we don't use basic hybrid automata as defined in Section 4.2.1. Therefore, when talking about hybrid automata, we always mean the hierarchical version described in Section 4.2.2. Of course, probabilistic hybrid automata can be applied to the hierarchical variant, so that PHHA are the second kind of automaton that we use in this work.

The most important context for hybrid automata is the experience acquisition in RoLL. By defining a skeleton automaton (only containing the mode hierarchy), we can structure and identify experiences. PHHA are a basis for giving the whole learning process a universal underpinning, for identifying learning problems, and integrating and using the learned functions. The functions $flow$ and *prob* are different models of the agent behavior that can be used during execution. For example, it would be interesting to know the probability with which a certain plan fails (the probability of the transition to the failure mode) or a general picture of what happens while the agent sets the table, for example the rate and order in which the pieces of tableware are moved, the time needed, how often cupboards are opened. All are aspects of the flow function, although taken together they are still only an approximation of the real $flow$ function.

4.3 Modeling the Program Execution

For specifying experiences in RoLL it is necessary to have a model of the control program and the environment, because these are the sources of experience data. This model provides a basis for the description of which experiences are wanted. In this section we set the formal basis for modeling the robot control program by showing how RPL programs

can be regarded as hybrid automata. The other language levels described in Chapter 3 are covered as well.

Beside the specification of experience, hybrid automata can also represent prediction models for the robot to use. In this case probabilistic hybrid automata are needed to draw a realistic picture of the agent behavior. This use of hybrid automata in the learning process is sketched briefly in Section 4.3.5. A complete description of the role of hybrid automata in the learning process can be found in Section 4.5.

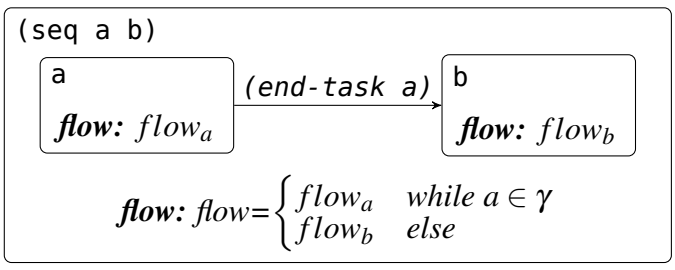
4.3.1 RPL Programs as Hybrid Automata

RPL programs can be modeled as hierarchical hybrid automata straightforwardly. Let $\mathscr{P} = \langle V, M, flow, T, act, cond \rangle$ be an automaton to model an arbitrary RPL plan. The set V is composed of two disjoint subsets W and F, where the first contains the normal variables and the former comprises all fluents. The modes correspond to the subtasks of a plan. The most primitive components of plans are LISP expressions that are represented by modes without submodes.

Transitions represent the relations of the subtasks. A task can have several subtasks when they are executed in parallel, sequentially or in conditional constructs like `if`. The *cond* and *act* functions depend on the construct that is to be regarded, see examples on pages 51 and 54.

The $flow$ function of LISP constructs describes the effects of its execution on the data state. The continuous effects of RPL constructs are composed of the effects of their sub-automata. Figure 4.3 illustrates this compositionality with the sequential composition statement `seq` of RPL. The sub-automata are chained in a way that when the first one has finished the second one starts. The flow function of the overall automaton first corresponds to the flow function of the first sub-automaton $flow_a$ and after the transition has occurred it is identical to $flow_b$.

Figure 4.3 Sequential composition in RPL as hybrid automaton.

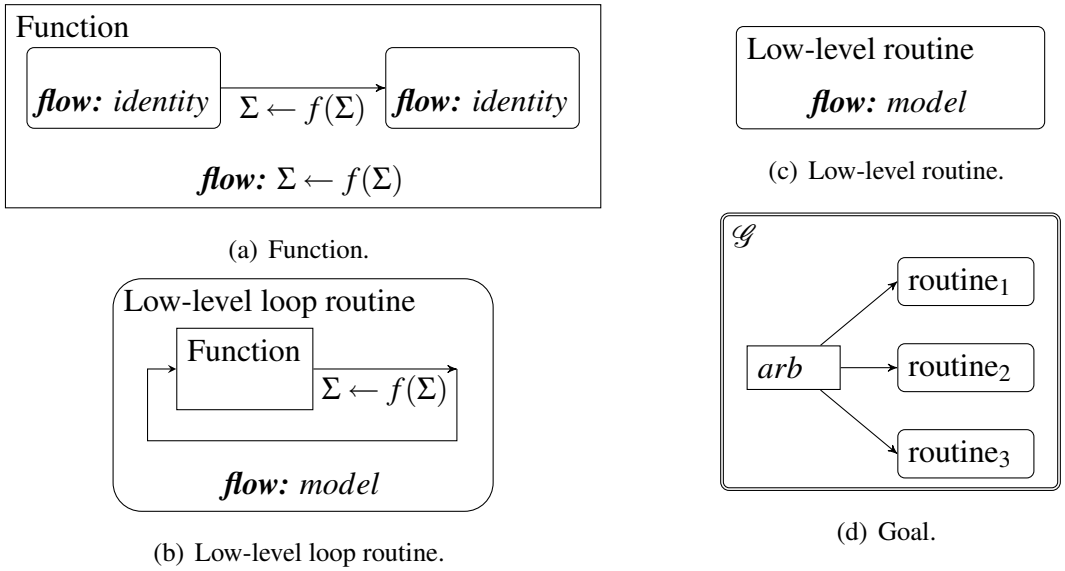


4.3.2 BDI Language Level

The BDI language presented in Section 3.3 on page 34 is implemented in RPL so that everything that can be expressed in this language can be modeled with hybrid automata. However, we have presented special control structures like goals and routines that make a more convenient notation appropriate.

Figure 4.4 shows a summary of the special control structures and therefore special automaton types encountered in the BDI language. Functions are automata that are executed instantaneously and change the data state in the specified way. Low-level routines can either be provided by an outside system or be implemented within the BDI language. In the first case the $flow$ function (or some aspects of it) can be learned from experience, but the inner structure is not known. Low-level routines are the most basic units of a control program. The overall flow function of the program can only be determined if the flow functions of the low-level routines are known, which should be relatively simple, because low-level routines only capture very primitive behavior. For a special kind of low-level routines the structure is known however. These are the ones that operate in a

Figure 4.4 Modeling of BDI language concepts with hybrid automata. The specific concepts are depicted in different ways as automata to distinguish them more clearly. Functions are henceforth depicted as boxes with sharp corners, low-level loop routines with widely rounded corners, and goals by a double line. Ordinary low-level routines are not distinguished by a special drawing mode, but can easily be discerned by not containing any sub-automata.



(a) Function.

(c) Low-level routine.

(b) Low-level loop routine.

(d) Goal.

single feedback loop, where the function setting the commands can either be learned or programmed. Here the structure is more open, but it is usually not enough for determining the flow function. The direct influences of the data state are given by the constant assignment of $\Sigma \leftarrow f(\Sigma)$, but the effects on the observable state variables caused by the execution of the routine in the environment are not known.

Goals are represented as data structures in the BDI language, but in order to achieve a goal the command `achieve` is executed. Its functionality is shown in the automaton structure in Figure 4.4(d). Depending on the judgment of the arbitration function, one of the possible routines that could reach the goal is chosen and executed.

We haven't mentioned goal routines here, although they are an important control concept in the BDI language level. The reason is that they can be treated as normal RPL procedures. Their $flow$ function is theoretically a result of the compositions provided by the RPL constructs and on the bottom the models of the low-level routines. In practice the flow function of higher-level goal routines should also be learned, because the composition becomes infeasible.

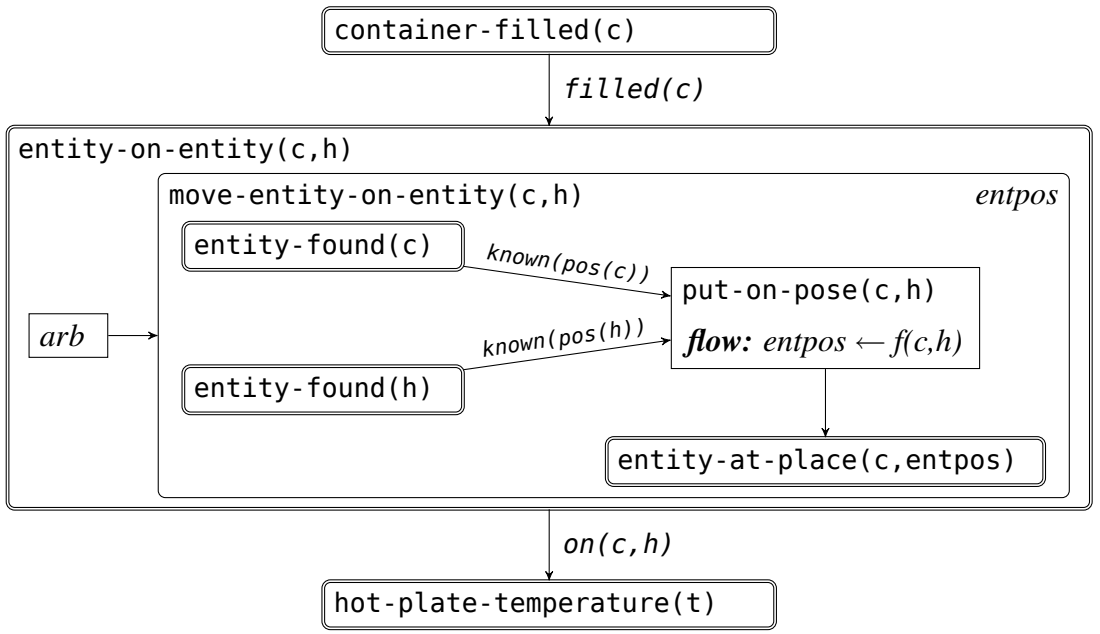
4.3.3 Modeling Programs for Experience Acquisition

Figure 4.5 shows a detailed example of how plans are modeled in the language underlying RoLL. The goal is to make water boil. The presented plan contains several subgoals: `container-filled`, `entity-on-entity` and `hot-plate-temperature`. All of these goals are achieved by an appropriate routine chosen by arbitration functions. The second plan step is modeled in more detail than the others. The routine `move-entity-on-entity` first has to find the objects needed and ascertain their identity. After recognizing the objects, a function `put-on-pose` determines a good position where the top object should be placed on the bottom object. After that the goal `entity-at-place` is established, whose execution is again modeled in detail.

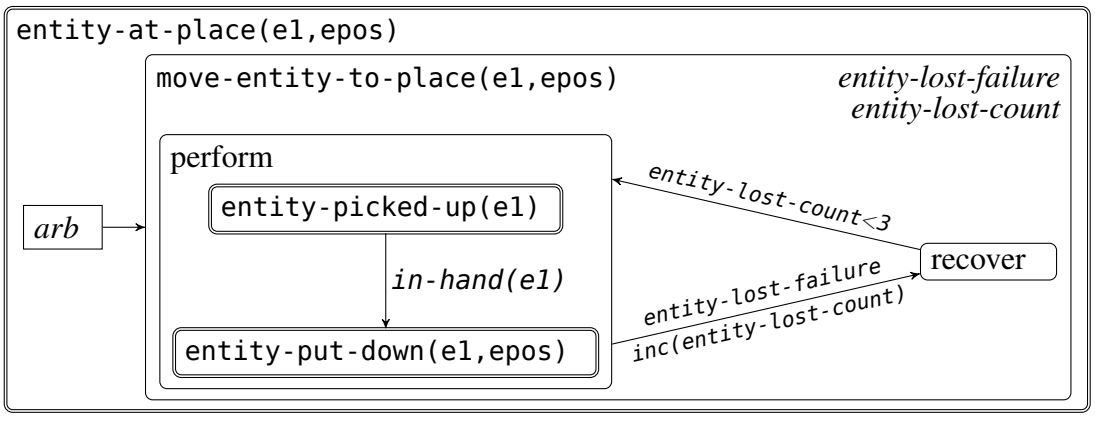
The routine `move-entity-to-place` demonstrates how failure recovery is modeled with hybrid automata. When an `entity-lost-failure` is detected (by a process which is not shown in the figure for reasons of simplicity), the automaton state shifts to the recovery mode. In this case the recovery simply consists of trying to move the object again. After three trials, no more attempts are made and the routine terminates with the status *failed*.

Knowing the structure of RPL programs in terms of hybrid automata is important for specifying experiences in RoLL. The automaton in Figure 4.5 is a non-probabilistic hierarchical hybrid automaton. In fact, for specifying experiences even the conditions and the transitions are superfluous, but they describe the working of the plan more clearly in the example.

Figure 4.5 Plan for making water boil modeled with hybrid automata.



(a) Top-level plan for boiling water.



(b) Subplan for placing entities.

4.3.4 Modeling the Environment with Hybrid Automata

Hybrid automata describing the program associate the automaton activity with processes in the control program. But for defining episodes, not only the internal program structure is important, but also the situation of the environment. However, the processes in the world are not as clearly defined as the program processes. For example, the time when a person is in the kitchen could be described as such a process. In contrast to the program process, we don't know about the person's intentions and must therefore define this episode by means of observation.

Therefore, the anchoring of the experience automaton to changes in the environment is defined by the concept of invariants. The classical definition of hybrid automata assigns invariant conditions to automaton modes. When the invariant of a mode is violated, a transition is activated. We have deliberately omitted this detail in the definition, because for most of our concepts we don't need invariants and invariants can be simulated by the condition function as shown in Figure 4.6. Instead of defining an invariant, the violation of the invariant is defined as a jump condition.

Defining environment automata by invariants means that a model automaton is identified only by a condition on the sensor data, which is given as an RPL fluent. As long as the condition holds the automaton is active, whereas the activation is stopped as soon as the condition becomes false.

This treatment of environment conditions in hybrid automata is much simpler than the definition of program automata. On the other hand, it is easy to understand and the concept of fluents allows arbitrarily complex expressions, so that any condition can be expressed. Besides, the invariants specifying outside conditions can be combined with the complex automata defining the program execution. Like program automata, the automata using invariants can be nested, which allows a fine-granular definition of episodes to observe.

Figure 4.6 Simulating invariants in hybrid automata.

(a) Automaton with invariant. A transition takes place when the invariant is violated.

(b) Equivalent automaton, where the invariant is simulated by a transition condition.

4.3.5 Prediction Models of the Control Program as Hybrid Automata

Hybrid automata are not only a means of specifying experiences. They can also be used as a framework for prediction models of the program. The modeling of the language layers

with HA presented in sections 4.3.1 and 4.3.2 makes some simplifying assumptions that are fine for modeling experiences, but are inadequate when predictions of the program behavior are needed.

As an instance, it would be useful for the robot to know the delays in switching from one action or sub-plan the next one. Figure 4.7(a) shows a hybrid automaton modeling the phenomenon that after one subprocess has terminated, the second doesn't start immediately, but some time elapses before it is activated using the following plan:

```
(seq
  (pursue action1
          (wait-for cond))
  action2)
```

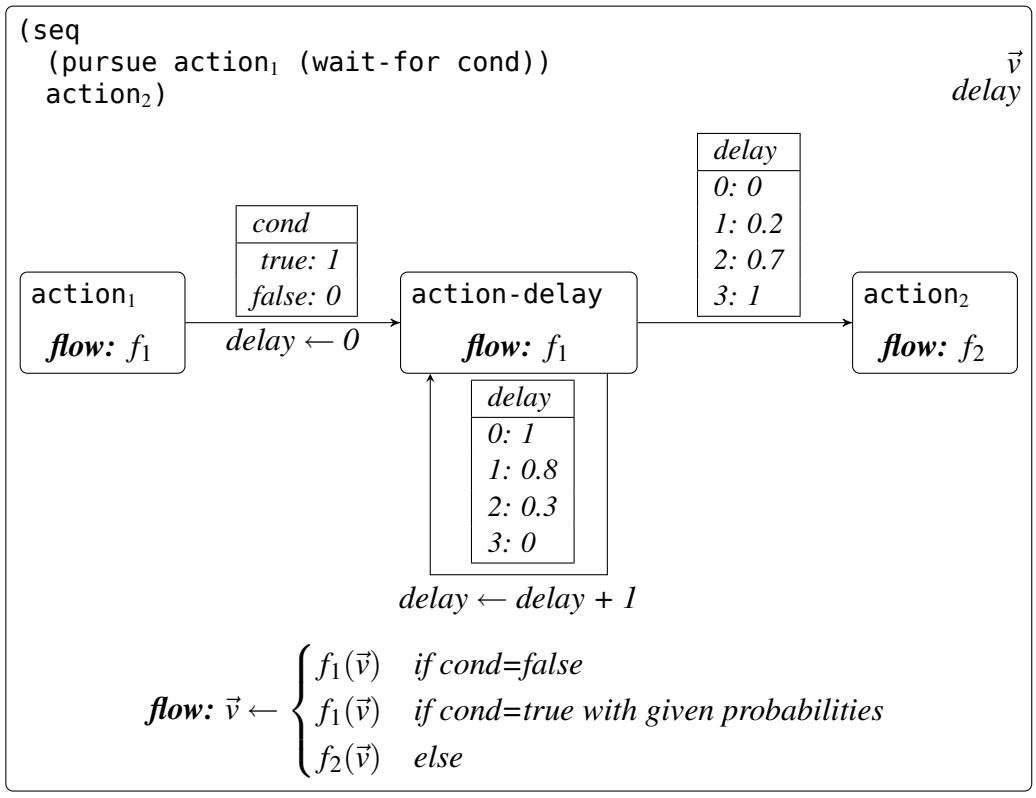
This plan consists of two sub-plans. The first, `action1` is executed until a certain condition `cond` holds. After that, `action2` is performed. Both actions are modeled by a mode containing their behavior, in this case as a black box view represented by the $flow$ function. Because the determination of the condition being fulfilled and the process switch both take time, the change of processes is not instantaneous. This fact is taken into account by the virtual mode `action-delay`. Its behavior is the same as that of `action1`, because the process change has not yet taken place. Depending on the given probability distribution, the delay lasts between one and three seconds.

The probability distribution of the transition from the `action-delay` mode to `action2` is not known a priori and should be determined empirically. Experience-based learning with RoLL can replenish this information. By the way, the automaton for modeling the agent program for the experience acquisition for this learning problem would look slightly different. The `action-delay` automaton cannot be sensed as such, so that the experience automaton would only contain the two action modes.

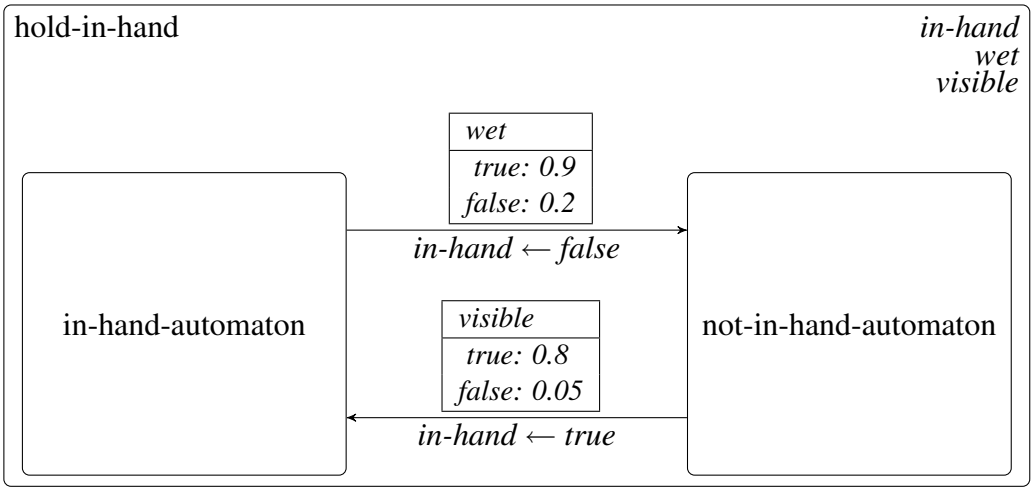
Probabilistic hierarchical hybrid automata cannot only model uncertainty produced by the program execution, which is in most cases predictable. A deeper source of uncertainty are of course environment processes. Figure 4.7(b) shows a model for situations when the agent loses objects that it has in its hand. The transitions from an automaton to itself are omitted in the figure. When there doesn't occur a transition to the other automaton, a transition to the currently active one is performed. In this way the postulation that the truth values sum up to 1 is fulfilled. This automaton is a representation of when objects slip from the robot's gripper. It can be used to forestall failures and make actions more reliable, for example by using both hands, when valuable or fragile things have to be carried.

The examples of this section should demonstrate how realistic robot behavior can be modeled with hybrid automata. The embedding in the learning process is described in more detail in Section 4.5. The following section takes up the use of hybrid modeling in the context of experience acquisition, which is an important prerequisite for understanding the RoLL constructs concerned with experiences.

Figure 4.7 Realistic modeling of robot behavior with probabilistic hybrid automata.



(a) Modeling the delay of state transitions in the program.



(b) Automaton for modeling the slipping and grasping of an object.

4.4 Experiences as Hybrid Automata

We have shown how the control program can be modeled by means of hybrid automata. In the following we describe the next steps of the experience acquisition and handling process. First, we define what an experience is in the context of the hybrid automaton model. Second, we lay the foundation for experience abstraction by introducing the concept of automaton transformations.

4.4.1 Experience Acquisition

Recall the definitions of the state of a hybrid automaton from Section 4.2. We differentiated between the data state σ containing the values of the variables in V, the program state γ representing the set of currently active automata in the hierarchy, and the automaton state θ as the combination of data and program state.

When we think in terms of programs, accessing the data state is straightforward, it is represented by variables, whereas the program state is usually not represented explicitly. Similarly, the concepts of episodes and experiences can be defined more straightforwardly for the data state than for the program state of automata.

Therefore, we map the program and consequently the whole automaton state to the data state. First we define the set of all automata contained in a hybrid automaton $\mathscr{H}$ as

$$A_{\mathscr{H}} = \mathscr{H} \cup \bigcup_{\mathscr{M} \in M_{\mathscr{H}}} A_{\mathscr{M}}$$

Now we extend the set of variables $V_{\mathscr{H}}$ with new variables $X_{\mathscr{H}} = \{v_a | a \in A_{\mathscr{H}}\}$. Each of these variables is mapped as a constant to its respective automaton. The program state now corresponds to the data state $\sigma_{X_{\mathscr{H}}} : X_{\mathscr{H}} \to A_{\mathscr{H}}, v_a \to a$ over the new variables. The complete automaton state[2] $\vartheta_{\mathscr{H}}$ is represented by the data state over the set of all variables $\sigma = \sigma_{V_{\mathscr{H}}} \cup \sigma_{X_{\mathscr{H}}}$.

This transformation of automaton to data state corresponds to the unique possibilities of RPL to represent its execution status within the program. An automaton without an internal representation of its program state models systems appropriately that don't know anything about what they are doing, like most computer programs or a robot that can set the table correctly, but when asked what it is doing is unable to tell you that it is trying to bring a cup to the table. For the acquisition of experiences it is important that the program state be represented in the variables, so that all internal aspects of the execution can be observed. For example, we might be interested in how often the robot fails when executing carrying tasks.

[2]We type θ for an automaton state represented as a tuple of data and program state and ϑ for an automaton state represented as a data state.

The automaton state changes constantly during execution. We call the mapping $\tau \rightarrow \vartheta$ from time points to automaton states during the run of automaton $\mathscr{H}$ an *episode e* of $\mathscr{H}$. It contains empirical evidence of how the automaton actually worked in contrast to the structural description of the automaton definition. If we compare hybrid automata to finite automata, an episode corresponds to the words accepted by the finite automaton. In the finite automata domain, a language can be defined either by an automaton or by a description of the language. In our case the language would be the set of all episodes through the hybrid automaton. Of course, we can only obtain finite sets of episodes, so that they can never be an unambiguous definition of the automaton, but they can be seen as an approximation to the real automaton behavior.

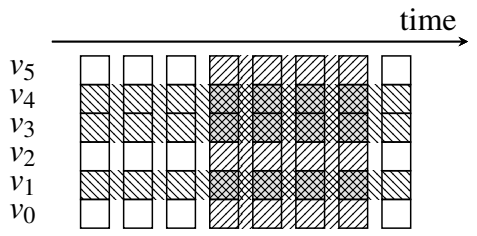
A complete episode of an automaton execution would be unmanageable for experience collection. We can limit the flood of information along two dimensions. The first we have already mentioned in Section 4.1, namely the partial definition of automata. Instead of modeling the complete program, we only model the sub-automata we are interested in. The other dimension is reduction of data by state projection. Given a set of variables V and a data state σ, we define a *projection* of a set $W \subseteq V$ on σ as $\sigma_W : W \rightarrow D, \sigma_W(v) = \sigma(v), v \in W$.

Figure 4.8 illustrates the experience acquisition in terms of episodes. It shows the variables $v_0 \ldots v_5$ of a partially specified automaton along the timeline. An experience (the crosshatched areas in Figure 4.8) is defined by an episode (the areas hatched from bottom left to top right) and a state projection (the areas hatched from top left to bottom right).

We have defined experiences abstractly in Section 1.2 on page 2. Now we can define a *raw experience* in more technical terms. The first component is a sketch of a hybrid automaton that corresponds to parts of the hybrid automaton represented by the control program and only contains the hierarchical structure of modes without specifying transitions or any of the functions of a hybrid automaton, thus specifying an episode to be observed. The second constituent of a raw experience is the data observed while the specified experience automaton is active.

We define the *activation period* of an automaton as being represented by a closed

Figure 4.8 Discrete illustration of an episode and state projection therein.



interval $[begin, end]$, whose first point is called *begin* and whose last point is called *end*. Often, the state at the beginning or end of an automaton run is of particular interest. We therefore allow — besides the normal episodes through an automaton — data acquisition events, where information is only recorded once. The data projection at these events can differ from the one of the trace that is captured in the interval during automaton execution.

4.4.2 Automaton Transformations

After having obtained raw experiences, these must be abstracted for facilitating the learning process. In the framework of hybrid automata, the abstraction corresponds to a transformation of automata.

The hierarchical structure of the hybrid automata we use is the basis of two types of abstractions. Either the structural representation of an automaton can be collapsed into a flow function or the other way round an automaton that is only characterized by its flow function can be expanded into subautomata. This is the natural way of moving through the hierarchy of automata. The second transformation of expanding an automaton can only be approximated or presumed, because if the structure of the automaton were known, the expansion would be unnecessary.

Another type of transformation is the already mentioned possibility of partial specification. An automaton can be transformed to a more abstract one by neglecting some details of the internal structure and simplifying it. This amounts to reducing the number of modes in the automaton. However, the remaining modes needn't correspond exactly to previous ones, but can be obtained by combining the purpose of two former modes into one. In the example of Figure 4.5(a) on page 51, the two modes `entity-found`, each ascertaining the location of one object, could be abstracted into one that finds all the entities needed for the task.

To summarize, an automaton transformation is a mapping from one HA $\mathscr{H}$ with a set of episodes $E_{\mathscr{H}}$ to HA $\mathscr{I}$ with episodes $E_{\mathscr{I}}$ using any of the three following operations:

- abstract: $\langle M_{\mathscr{H}}, E_{\mathscr{H}} \rangle \rightarrow \langle flow_{\mathscr{I}}, E_{\mathscr{I}} \rangle$
- expand: $\langle flow_{\mathscr{H}}, E_{\mathscr{H}} \rangle \rightarrow \langle M_{\mathscr{I}}, E_{\mathscr{I}} \rangle$
- restrict: $\langle M_{\mathscr{H}}, E_{\mathscr{H}} \rangle \rightarrow \langle M_{\mathscr{I}}, E_{\mathscr{I}} \rangle$

The episodes must be adapted according to the structural transformations.

The definitions just presented should serve as a reference for the next chapter, where we present the RoLL constructs for defining and abstracting experiences. Their syntax is built along the lines of the hybrid automaton model.

4.5 The Learning Process

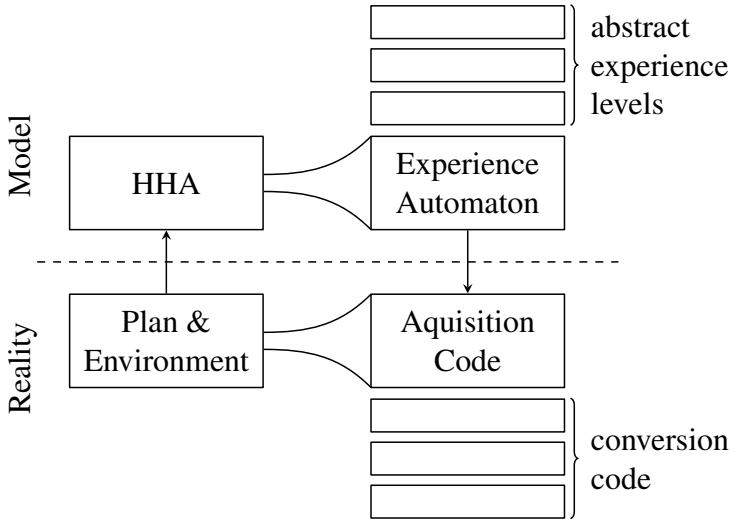
In this chapter we have presented the theoretical background of RoLL. We have sketched several aspects of hybrid automata in the learning process. Now we give a summary of these concepts and draw a complete picture of the learning process from the viewpoint of hybrid automata.

Let us recall the steps that are necessary for learning from Figure 2.2 on page 19: experience acquisition (possibly guided by a problem generator), abstraction and management, the learning process and embedding the learned routine into the control program.

The first part of this process is depicted in Figure 4.9. The robot control program corresponds to a huge hybrid automaton, which integrates not only the working of the program, but also its execution in the environment. For specifying the wanted experience, a partial model of the control program is needed, the experience automaton. It is specified in the form of a hierarchical hybrid automaton, which is anchored to the control program, so that the transitions in the automaton are triggered by events (i.e. changes in variable values) in the program. The declarative specification of an experience by the experience automaton is translated back to executable code by RoLL. When performed in parallel to the primary control program it produces a data structure containing the desired experience data.

In the domain of hybrid automata, the experience automaton can be abstracted as described in Section 4.4.2. This produces other automata representing different, more abstract aspects of the experience automaton. The transformations specified on automata are used by RoLL to produce code that automatically converts raw experiences to abstract

Figure 4.9 Relation of hybrid modeling to program execution and experience acquisition.



ones and stores them as desired by the programmer.

In its current version, this is the scope of where RoLL makes explicit use of hybrid automata. But the rest of the learning process can be considered in the light of hybrid automata as well. Figure 4.1 on page 41 illustrates this procedure.

We have provided examples of realistic models of parts of the control program in Section 4.3.5 on page 52. By modeling the structure of the agent program, we would have a universal representation of models of the robot's behavior in the world. On the one hand this representation would allow a simple, integrated access to models. On the other hand, it would enable a uniform way of identifying problems to be learned. For example, when a plan is to be tested for possible flaws, the robot applies performance models of the routines needed in this plan. Because of the universal accessibility there is no need to know if the model is present or not. It is worth while to give it a try. If there is no such model, it is a good candidate to be scheduled for learning. If it already exists, so much the better, it can be used immediately.

The learning problem as such gains a lot of meaning if the function that is to be learned is defined in the light of hybrid automata and the learned function is accessed in this context, too. By knowing if a low-level routine, an arbitration function or a prediction model is to be learned, the integration can be performed more smoothly and universally. Besides, this knowledge is useful for the problem generator. A model needs other kinds of experiences than a routine. By coordinating the problem generator and critic element, the parts of the state space for which there is not enough experience collected yet can be explored in a target-oriented way.

4.6 Summary

We have defined several variants of hybrid automata, the most important for our work being hierarchical hybrid automata. The theory of hybrid automata serves as a formalism to define experiences, experience abstractions and learning problems.

For specifying raw experiences the execution of the robot program in the world is modeled by hybrid automata and anchored to the control program, so that the automaton is activated according to its definition. The data to be observed is defined as a projection on the detected episode.

Experience abstraction can be viewed as a transformation of hybrid automata. This allows a uniform syntax for all kinds of experiences and their conversions.

Finally, we have pointed at a wider scope, where hybrid automata appear in the learning process. When the learning problems are defined in terms of hybrid automata, their semantics is clear to the program, so that an automatic choice and specification of learning problems becomes possible.

4.7 Related Work on Hybrid Modeling in Autonomous Systems

Hybrid automata have been used for a variety of purposes in the context of model-based robot control, planning and languages for autonomous robots.

Beetz and Grosskreutz (2005) model a control program with hybrid automata, in order to make predictions about plan execution scenarios. This makes it possible to forestall plan failures and to compare different plans for achieving a certain goal. This method of plan projection is especially relevant for transformational planning in real-world environments, where the plans cannot be analyzed without knowledge of the environment. Accurate models are necessary to predict the execution result of the plans.

For implementing systems that require a high degree of autonomy and reliability, the program should contain declarations of what state the system should be in rather than a predefined sequence of actions (Williams and Nayak 1996b; Williams et al. 2003). When such a program is equipped with models about the underlying hardware, model-based programs are much more robust than traditional ones, because they recognize hardware failures and can choose between several solutions that achieve the same system state. This allows fast reconfiguration when hardware failures are detected. Since robustness and autonomy are crucial factors for space exploration projects, model-based programming has been applied in several ones like Remote Agent (Muscettola et al. 1998), Livingstone (Williams and Nayak 1996a), the Mission Data System Project (Volpe and Peters 2003) and others (Barrett 2003; Ingham, Ragno, and Williams 2001; Knight, Chien, and Rabideau 2001). These systems control the hardware directly and have thereby shown impressive reliability. However, for more complicated robotic applications that are to fulfill higher-level tasks in real-world environments, this approach cannot be carried over directly, because the search space of finding configurations (or plans) for achieving certain goals gets larger. On the other hand, the execution of high-level robot plans can be enhanced significantly by using models of the robot's behavior (Fox et al. 2006; Infantes, Ingrand, and Ghallab 2006). This is one of the main motivations for RoLL, to learn models of all parts of the robot program and update them continually.

Hybrid automata have also been used by Fox and Long (2006) for designing the plan language PDDL+. PDDL is the language used in the annual planning competitions (McDermott 2000). Therefore, it is developed further regularly for integrating more sophisticated concepts into the planning language and make the planning tasks more challenging. Fox and Long (2006) extend PDDL in a way that actions needn't be instantaneous as assumed by most planning systems, but can have a longer duration. In contrast to our work, PDDL+ doesn't only make use of the well-understood framework for modeling continuous actions, but also uses reachability findings for proving the existence of plans in a certain domain.

Chapter 5

Robot Learning Language

With the background from the last two chapters on the underlying language levels and the theory of hybrid automata, it is now time to present RoLL, the Robot Learning Language. We first give a brief overview of the steps involved in specifying and executing a learning problem. The most crucial part of RoLL in its current form are experiences, which have already come up several times. In this chapter we show the RoLL mechanisms for collecting, abstracting, storing and managing experiences. Of course, the learning process as such plays an important role, too. We will explain, what kinds of functions can be learned with RoLL, how different learning systems can be used, and how the result of the learning process is integrated into the control program. The main focus of this work is to describe the concepts of RoLL, not its syntax, which is however apparent in the examples. A reference of the RoLL syntax is provided in Appendix C and a complete code example is presented in the next chapter.

5.1 Experience-based Learning

RoLL is designed to support any kind of learning involving data that a robot can acquire by itself. This is what we call *experience-based learning*, which is the most common method of learning on real-world robotic systems. It does not include the enhancement of knowledge by pure logical deduction for instance. In the following we structure experience-based learning methods according to several criteria and sketch their support in RoLL. Then we give an overview of how learning is performed and the main parts of the language.

5.1.1 Classification of Learning Problems

Learning problems can be classified along several dimensions. One common criterion is the differentiation of supervised versus unsupervised learning. In the supervised learning paradigm the agent is provided with experience in the form of input-output pairs from a teacher. In contrast, the output is not given in unsupervised learning. The task here is to find correlations in the set of input data. RoLL doesn't differentiate between supervised and unsupervised learning. The experience acquisition works in both scenarios identically. However, we assume that experiences for supervised learning are not provided by a teacher, but are acquired by the robot itself, which is a very realistic assumption for robot learning.

Another way to differentiate classes of learning problems are the online and offline learning paradigms. Offline learning takes place before the function is needed in the program. This means that first all required experiences are collected, then they are processed by a learning system and integrated permanently into the control program. Online learning takes place while the robot is performing its work. RoLL supports both online and offline learning, because it separates the experience acquisition from the learning process and thus allows arbitrary interaction between the primary program and the learning part. The rate of online- or offline-ness can be adapted, so that the border between those two paradigms is not fixed.

Somehow related to the question of online and offline learning is the issue of active versus passive experience acquisition. An integral feature of reinforcement learning and other online techniques is action selection while learning. This means that the robot chooses actions that lead to useful experience or maximum reward. The task of determining these actions is performed by the problem generator element of the architecture in Figure 2.1 on page 17. In the current implementation of RoLL this issue is not supported to its full extend. We only provide means for simple a priori problem generation (see Section 5.2.3), which corresponds to the state of the art in current robotic systems. The alternative way for getting experiences is simply watching the robot while it is doing its normal activity. In this passive scenario, the critic element only watches what is going on and uses this information for later learning.

The learning in online/offline mode with active and passive experience acquisition is pictured in Figure 5.1. We assume that the robot has times where it doesn't have to fulfill any jobs given by the user. The middle column depicts the robot's busy and free time. The columns on the left and right illustrate how different learning modes interact with the primary activity. Active periods can be used for passive experience acquisition. When there is free time, this experience is used for learning.

An alternative would be to wait for a leisure time slot and then explore interesting parts of the state space actively. The active exploration can either be used for offline learning or in the context of online learning, where the succession of experience collection and

learning is very fast. In contrast to the offline learning paradigm, where all actions aim at acquiring experiences, the problem generation for online learning normally takes into account the robot's goals to be achieved as well as the need for useful experiences.

The figure shows offline and online learning in their pure form, but this is not necessary when working with RoLL. The offline learning task with passive experience acquisition could be continued in the next cycle of normal robot activity by collecting more experiences. This procedure can be repeated until the learning result is satisfactory.

5.1.2 Learning Process

The process of learning in RoLL is similar to that described by psychologists. Kolb (1984) distinguishes four phases in human learning, which are repeated cyclically:

1. Concrete Experience
2. Reflective Observation
3. Abstract Conceptualization
4. Active Experimentation

After making an observation, we reflect on the possible implications of this observation. Then the observation is regarded in the context of previous experiences and knowledge (abstract conceptualization). Finally, the conclusions drawn from this process are tested by active experimentation, which provides new experience. By repeating this cycle, experiential learning is a continual process.

Figure 5.1 Integration of different learning paradigms in the normal control flow.

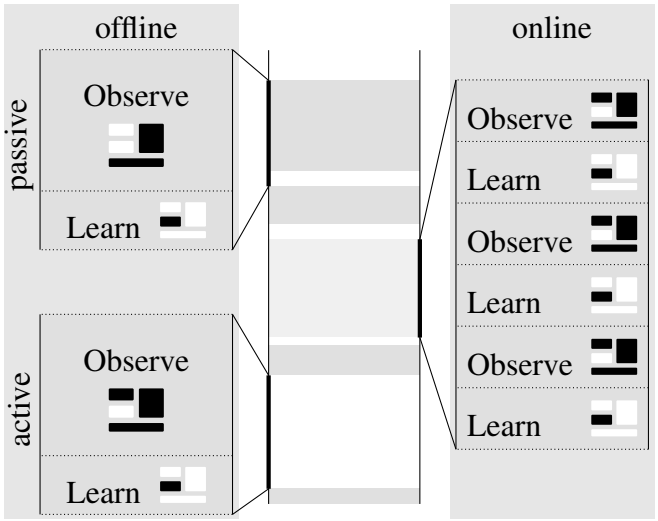


Figure 5.2 shows the learning cycle concept in RoLL. The basic version depicted in Figure 5.2(a) contains only two steps: an experience gathering step and a learning step. Viewed in Kolb's model, the experience gathering step comprises the first two steps of the human experiential learning cycle, because it includes not only the recording of observations, but also the abstraction process for generating more meaningful experiences for learning. The abstract conceptualization step corresponds to the learning step in RoLL. The new experiences are used for modifying the program according to the observations. The step of active experimentation is a natural step when the program is modified, every subsequent action can be viewed as an experimentation with the newly learned function. Viewed in a narrower sense, active experimentation can comprise reflection of what steps to take next in order to observe useful experience and is therefore part of the RoLL experience gathering step. This sophisticated functionality of the problem generator is not implemented to great depth in RoLL, but it can be added for specific problems. On the other hand, active exploration is not always necessary, as Polly B. Berends puts it:

> Everything that happens to you is your teacher. The secret is to learn to sit at the feet of your own life and be taught by it.

For humans it is natural to incorporate newly observed experiences at once. However, the running of a learning algorithms can be quite slow and even inappropriate, as many learning algorithms need a lot of training data to work properly. Therefore, the experience gathering step in RoLL can comprise the observation of several experiences before the learning step is started. Sometimes, a complete learning problem can be solved in just one cycle, which is in fact the current approach for neural network learning on robots: acquire lots of data and then use it for training a neural net.

Learning in several cycles is particularly important for online learning algorithms like reinforcement learning. In this case, the execution time of a single cycle is very short, often only gathering one new experience. The learning takes much more cycles than in offline learning scenarios.

Although people learn constantly and never really stop improving, at some point the learning of a specific aspect slows down significantly, because it is already performed skillfully. An example are basic everyday motor skills, which are hardly advanced after childhood. In robot learning, an analogous process should take place. When some skill has been learned satisfactorily, the learning should be stopped or run at larger intervals, so that resources are available for other learning problems.

In order to decide when a problem has been learned to a sufficient level, the learning process itself can be observed and evaluated. One way of doing this is to expand the abstract conceptualization step to not only draw conclusions from the observations in the form of learning results, but also to evaluate these results by comparing the experiences that were used for learning with the result of the learning process. Figure 5.2(b) illus-

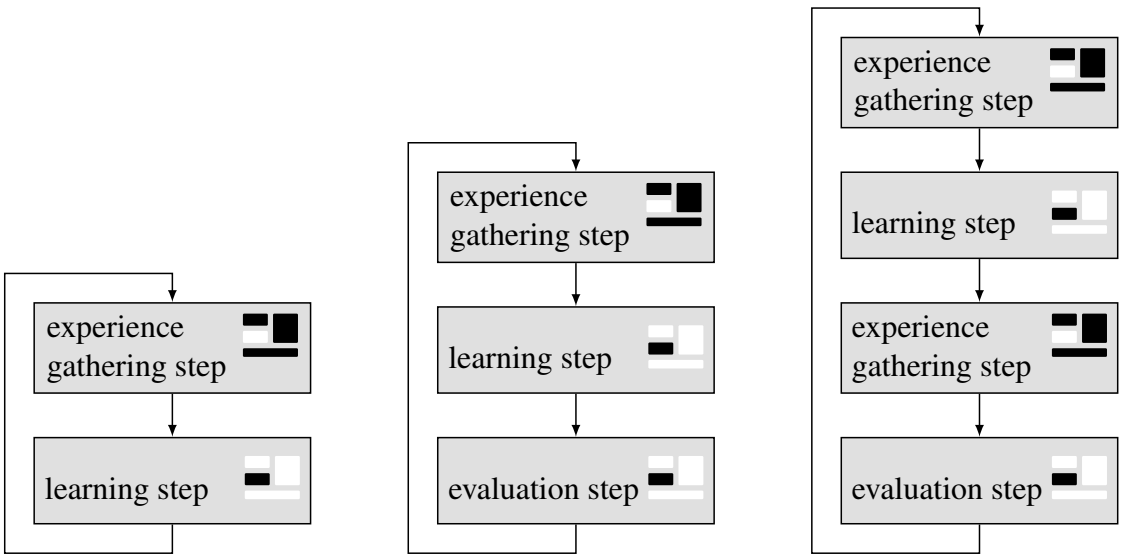
trates this procedure. An example of this kind of evaluation is cross validation in neural networks.

Another way of evaluating the learning results is to observe the robot using it and evaluate its performance. In this case (see Figure 5.2(c)), the complete process consists of two cycles after Kolb: the active experimentation step of the first cycle takes care to produce good observations for the evaluation step, which works analogous to the learning step, replacing the learning system by an evaluation system and the function integration by feedback on the learning result.

Let us now have a closer look at what happens in the RoLL learning process, which is depicted in Figure 5.3. The right-hand side shows a RoLL program. It consists mainly of declarative definitions, some of which are applicable for several learning problems, some for a specific problem. The control program itself is only a small part of the overall program. The figure shows a very simple program where one learning problem is learned while the robot is performing its normal top-level plan, so that we have a case of passive experience acquisition.

There are two commands that make the program learn, whose functionality roughly corresponds to the learning steps in Figure 5.2(a). The left-hand side of Figure 5.3 depicts what these commands do. The `acquire-experiences` construct is run in parallel to the normal control program and observes desired data. In the perspective of our learning

Figure 5.2 Learning cycle of experience-based learning.



(a) Basic learning cycle.

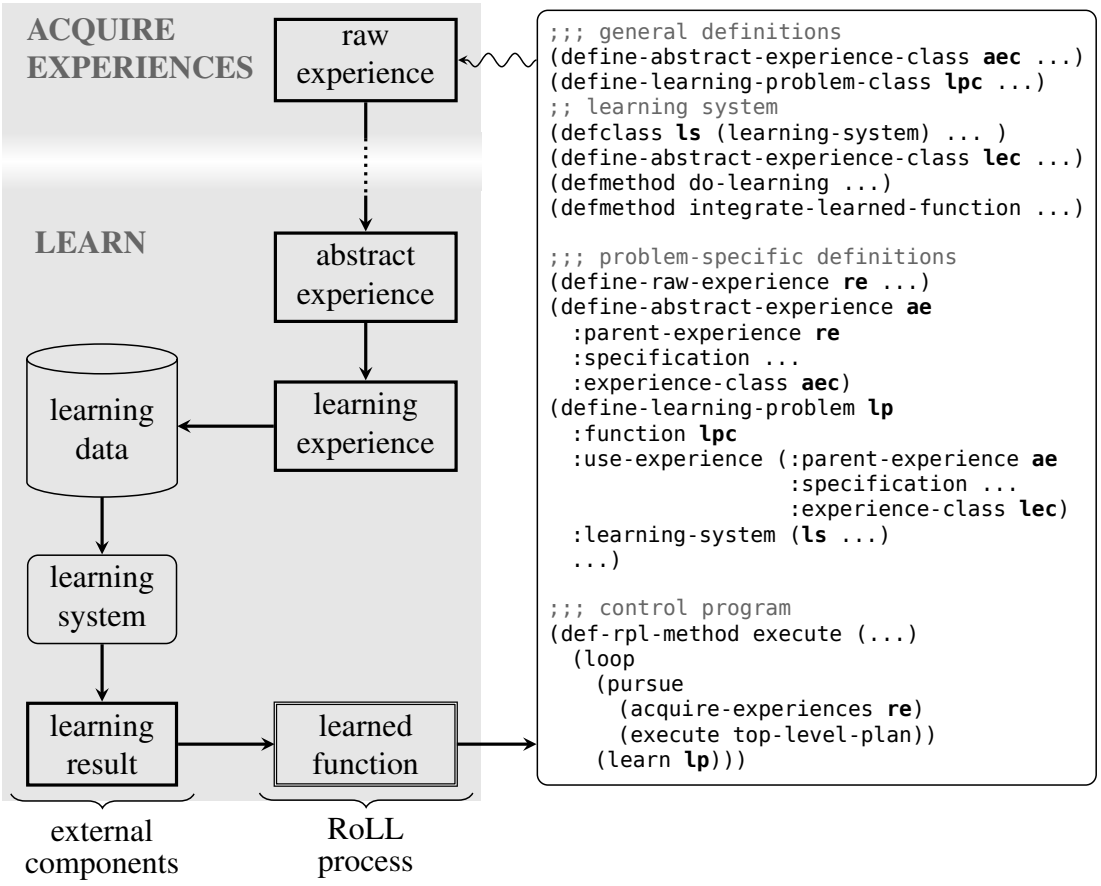
(b) Learning cycle with evaluation.

(c) Learning cycle with evaluation based on experience.

architecture from Figure 2.1 on page 17 this involves the critic element, whose job is to detect relevant episodes. For active experience acquisition the top-level plan would have to be replaced by problem-generating code setting parameters in the actual control program.

The logical experience gathering step of Figure 5.2 usually comprises more than what `acquire-experiences` does, because the observed data is only abstracted until it is written to a permanent storage device (see Section 5.2.1 for a detailed explanation). The last stages of abstraction are performed when calling the command `learn`, but this is just an implementational detail. The logical learning step starts after the learning experience has been generated. In the definition of a learning problem, the experience gathering step

Figure 5.3 The learning process in RoLL. The right-hand side shows a typical RoLL program, whereas on the left the steps of the learning process involving the commands `acquire-experiences` and `learn` are illustrated. The functional elements marked as "external components" are usually provided by external programs, but can also be implemented within RoLL.



is described by the `define-raw-experience` and `define-abstract-experience` constructs.

The learning step is specified with `define-learning-problem`, which describes what kind of function is to be learned and selects the learning system and an appropriate bias. This information is used when the command `learn` is called from the control program. Usually the learning takes place outside the RoLL program by using external learning software. It consists of transforming the abstracted experience to a form that is readable for the learning algorithms, the parameterization of and call to the learning software, and a transformation of the output obtained from the learning system to a RoLL function. The definition of the learning system and an interface for calling it is an example of the general specifications applying to several learning problems.

The simple program in Figure 5.3 doesn't comprise an evaluation step, but it would work in the same way as the learning steps. It usually involves some kind of experiences, either the ones that were used for learning or newly acquired ones. In both cases the data must be prepared in a way that an evaluation system can use it. Evaluation systems can be functions performing statistical calculations or even learning systems like decision trees that can classify the cases in which the learned function works well and when it has problems. The evaluation can also be included in the primary learning system, for example error statistics or cross validation in neural networks. Other than the learning result, the

Table 5.1 Summary of RoLL constructs with references to explanations, examples and specifications.

Concept	Language Construct	Reference
Problem-specific Constructs		
raw experience	`define-raw-experience`	5.2.2 (74) / 6.2.1 (104) / C.2 (145)
	`problem-parameters`	5.2.3 (82) / 6.2.1 (106) / C.3 (149)
	`with-problem-parameters`	5.2.3 (82) / 6.2.1 (106) / C.3.2 (149)
	`acquire-experiences`	5.2.2 (81) / 6.2.1 (106)
abstract experience	`define-abstract-experience`	5.2.4 (87) / 6.2.2 (107) / C.4 (152)
	`define-experience-conversion`	5.2.4 (90) / 6.2.2 (107) / C.4.2 (153)
learning	`define-learning-problem`	5.3.3 (96) / 6.2.3 (108) / C.5 (154)
	`learn`	5.3.3 (96) / 6.2.3 (108)
Problem-independent Constructs		
experiences	`define-experience-class`	5.2.4 (91) / C.6.1 (157) / D.1 (158)
learning	`define-learning-problem-class`	5.3.1 (94) / C.6.2 (157) / D.2 (162)
	learning system definition	5.3.2 (94) / C.6.3 (157) / D.3 (163)

judgment of the evaluation system is not integrated permanently, but used for deciding if the learning should be continued. If the evaluation provides enough information, this can also be used for the problem generator to know which regions of the state space should be explored in more detail.

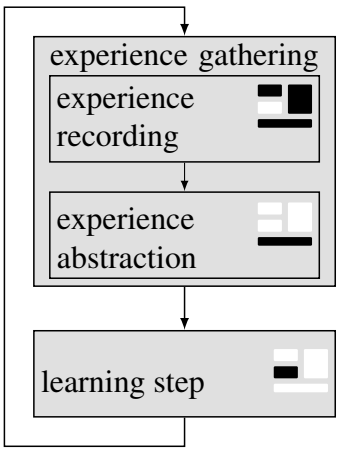
5.1.3 RoLL Constructs

Table 5.1 lists the language constructs defined in RoLL. As an guideline they are divided into general and problem-specific constructs and they are related to more abstract concepts. As a reference, the relevant sections in this chapter, examples in the next chapter or Appendix D, and a link to the syntax specification in Appendix C are provided.

5.2 Experiences

Experiences play a central role in the learning process. Figure 5.4 shows a detailed view of the experience acquisition step in the learning cycle. First, experiences must be observed while the robot is acting. This involves the performance element for the robot activity, the critic element for the observations, and possibly the problem generator for instructions how to control the robot. After a useful observation has been made, the experience is abstracted and stored by the critic element. In RoLL, an experience is a data structure that can be created and processed by different operations on it. Before describing the two stages of the experience gathering step we give an overview of the experience data structure and its subclasses.

Figure 5.4 Experience acquisition in the learning cycle.



5.2.1 Experience Classes

Experiences are represented as a data structure in RoLL. Its definition is based on hierarchical hybrid automata. Recall that an experience is a trace through a hybrid automaton observed during an episode, which is specified by a hybrid automaton. This means that the data structure specifying an experience must contain (1) the automaton structure and (2) a specification of the desired data based on the automaton definition.

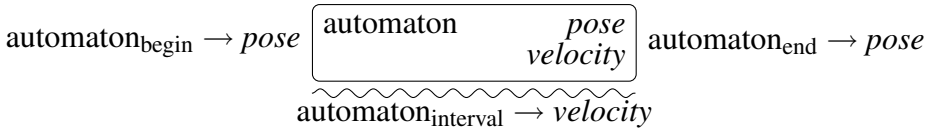
The automaton structure in experiences only contains modes and their relationship in the hierarchy. Transitions can be observed and deducted from the experience, but they are not specified in the structure. The modes of an automaton are supposed to run in parallel. The automaton structure can correspond to the task structure of the control program, to some hierarchical description of external events or an arbitrary logical conceptualization (see also Sections 5.2.2 and 5.2.4).

Hybrid automata, and therefore episodes are specified in a LISP class. The data observed during one episode is represented as an instance of such a class.

Data Addressing

The data of an experience is attributed to the (sub-)automata of this experience. Each automaton can contain data for the two events when the automaton starts and ends, and it can comprise a stream of data that corresponds to the time while the automaton is active. Figure 5.5 illustrates these data sources in experiences. The distribution of data in one

Figure 5.5 Illustration of experience data types. Data can be recorded once at the beginning and end of the automaton execution or continuously in between.



(a) Illustration of experience data. The example contains the robot's pose at the beginning and end of the automaton activity, as well as a stream of its velocity during the activation time of the automaton.

(b) Graphical illustration used from now on. In the automaton we mark that something is to be recorded at the beginning or end of an automaton by small boxes and the recording of interval data by filling the automaton box with gray.

automaton works also in the hierarchy of nested automata. This means that all kinds of events and arbitrary periods of time can be indicated by a skillful choice of the automaton structure. Figure 5.12 on page 88 shows an example for data specification in a nested automaton.

The data in an experience is addressed in a uniform way, no matter if the purpose is to write into the structure (as is done during raw experience acquisition) or to read data from it. For addressing a piece of data one has to specify (1) the automaton mode it is associated with, (2) one of the events *begin* or *end*, or the *interval* slot of the chosen mode, and (3) the name of the parameter. The specification `(:var pose (:begin automaton))`, for instance, returns the value of the pose slot in the begin event of the mode `automaton`.

There are some more details to the retrieval of data from an experience. First, intervals are data streams, which means that we must provide options for getting the whole stream as a list or single data points, for example the first value of the stream. This is done by providing additional keywords like `:all-instances` for the whole list or `:first-instance` for getting only the first value from the stream. The default is to return all instances, because this is the original motivation of recording interval data. For only getting certain instances, one could have defined events instead. Another dimension for ambiguity lies in the fact that modes can be activated several times during data gathering. Since only the hierarchical structure of the automaton is given, it can happen that some modes are activated more often than others, we don't assume any linear execution. If an automaton is activated several times, the data slots contain multiple occurrences of the automaton run. With the keyword `:all-occurrences`, all of them can be addressed as a list of values. With `:only-occurrence`, which is the default, an assumption is made that the automaton is only activated once and therefore only contains one occurrence. If several occurrences are detected, a warning is returned. Besides, single occurrences can be addressed by a number specifying the n-th occurrence of the experience.

To summarize, a value in an experience can be addressed by specifying

- the automaton name,
- the data source: *begin*, *end* or *interval*,
- the variable name,
- optionally the desired occurrence and instance.

The syntax of specifying and addressing experiences is listed in Appendix C. In the following we show some classification criteria of experiences along several dimensions: their degree of abstraction, the time of their existence and how they are stored.

Stages of Abstraction

We have already mentioned raw and abstract experiences. Raw experiences are the ones that are created when experiences are collected. They contain the data from state vari-

ables and local variables. For facilitating learning, often a more abstract representation is needed.

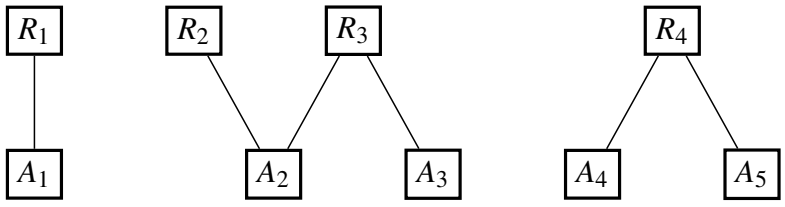
The first abstraction step can take place at the time the experience is acquired. For example, if you need the robot's distance to some fixed point instead of its absolute position, you can calculate this value when collecting the raw experience and only store the abstracted value. Other things cannot be computed at the time of the experience acquisition. For example, if the duration of an action is needed, you would define an automaton corresponding to the action and observe the global time variable at the *begin* and *end* events. For calculating the duration, you need both values, but at the time you record the time at the beginning of the automaton execution you don't know the second value yet. When recording the end time, the starting time is already recorded and cannot be deleted from the experience any more. In this case the abstraction must take place at a later time.

There are other reasons why experiences should be collected in a rather system-near way and be abstracted later. Often, raw experiences can be used for several abstract ones. For example, when we want to learn models of a navigation routine, we would like a model predicting the time needed for a task and one for forecasting possible failures. For both learning problems, similar experiences are needed. In both cases, the robot's original and goal position are of interest, only the output values of the functions to be learned differ. Here it would be advisable to collect raw experiences that contain the necessary information for both learning problems and later abstract them into two different abstract experiences.

Figure 5.6 shows that abstract experiences can also be generated from several kinds of raw experiences. This doesn't mean that the abstract experience is calculated from two instances of raw experiences, but that from any instance of any of the two raw experience classes R_2 and R_3 an abstract experience of type A_2 can be generated. An example would be to learn a model of the robot's navigation capabilities when it has several routines for navigation. There could be raw experiences for each navigation routine protocolling the time needed for the task. Each of these experiences is valuable for the abstract experience, so that the number of abstract experiences is the sum of the number of all instances of the raw experience classes.

In both cases — abstracting a raw experience into several abstract ones and getting ab-

Figure 5.6 Experience abstraction network.



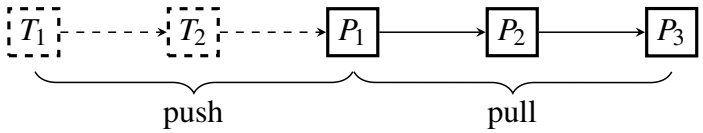
stract experiences from several raw experience classes — the raw experience acquisition could in theory be modeled in a way that there is a one-to-one relationship between raw and abstract experience. For example, we could watch the robot every time it navigates instead of protocolling the behavior of the particular navigation routines. But often, several things should be learned and the abstraction network provides an opportunity not to record redundant data and use the raw experiences economically.

Up to now we have talked about raw and abstract experiences in a way that suggests that there are only two stages of abstraction. This is not the case in RoLL. There can be an arbitrary number of abstraction steps involved until the experience is used for learning. Usually, there are about three stages: the raw experience, an intermediate level for permanent storage and a preparation for the learning step as shown in Figure 5.13 on page 92. In online learning mode there might be only one or two steps of abstraction, because the experience is not stored permanently. In cases where the experiences are used for lots of learning problems, there might be more intermediate steps. The structure of the experience abstraction network is a matter of design involving issues such as the number of learning problems and their relationships as well as storage capacities.

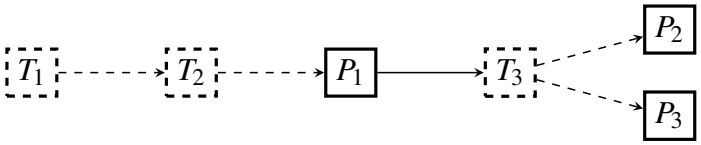
Volatility of Experiences

Because there can be many processing stages and not all experiences are needed permanently, experience classes can be differentiated as being *permanent* or *transient*. Permanent experiences are stored in a database or a file, so that they can be retrieved later. Transient experiences only exist long enough to pass their data on to the next experience processing step. After that they can be filled with new values. This is usually the case for

Figure 5.7 Processing chain of transient and permanent experiences.



(a) Typical processing chain starting with transient experiences and ending with permanent ones.



(b) Processing chain with a transient experience in a later processing step.

raw experiences. When the activation of the specified automaton ends, the experience is immediately transformed to the next abstraction step. After that, the values are overwritten by new ones of the next episode.

The different nature of permanent and transient experiences requires different strategies of processing as depicted in Figure 5.7. The transformation of transient experiences works in a push principle. Because the transient experience must prepare for new data, it has to get rid of the current values. Therefore, as soon as a transient experience is complete it calls the transformation function to produce the next step in the abstraction chain. In contrast, permanent experiences are stored without any knowledge of how they are to be processed further. Here we implemented a pull principle, so that the desired set of experiences is provided on demand, for example when the learning is to start.

Usually the first processing steps involve transient experiences and the later ones work on permanently stored experiences as shown in Figure 5.7(a). The number of steps depicted there is rather high, they are only necessary in complex abstraction networks. Figure 5.7(b) shows an example where transient experiences can also make sense in later processing steps, namely when a common abstraction step leads to different permanent abstractions. Instead of coding abstraction steps twice, they can be performed in one transformation, which makes the abstraction process more modular.

Good learning performance is often not achieved with lots of experiences, but with meaningful experiences, or as John Dewey put it:

> The belief that all genuine education comes about through experience does not mean that all experiences are genuinely or equally educative.
>
> (Dewey 1938)

Therefore it is advisable to filter experiences in the course of abstraction. In the case of transient experiences this is rather difficult, because only one experience is given at a time. In contrast, permanent experiences provide an opportunity to extract only a subset of the most meaningful experiences. This selection process can be very complex and we haven't investigated much in this direction. But the permanent storage is a good starting point for sophisticated management of experiences. This can include temporal restrictions, e.g. that recent experiences are more reliable than older ones, or restrictions on the state space distribution.

Storage

Permanent experiences can be stored in several ways, for example in text files or in a database. Files are a very simple and compact way of storing experiences, but the retrieval can be difficult, if there is a complex parsing procedure needed. Besides, the experiences cannot be addressed separately.

A more sophisticated storage is provided by databases, where experiences cannot only be retrieved separately and by a standardized interface, but can also be manipulated and filtered with data mining mechanisms. Because learning often requires great amounts of data, a good abstraction step should be chosen for the storage in a database so that not all raw data sets are stored.

The RoLL core language doesn't impose a certain form of storage. By defining an experience class as a permanent experience and thereby specifying how experience is written and retrieved any kind of storage can be used. The default for permanent experiences is the storage in a relational database with a determined database schema, which is described in detail in Appendix D.1.2.

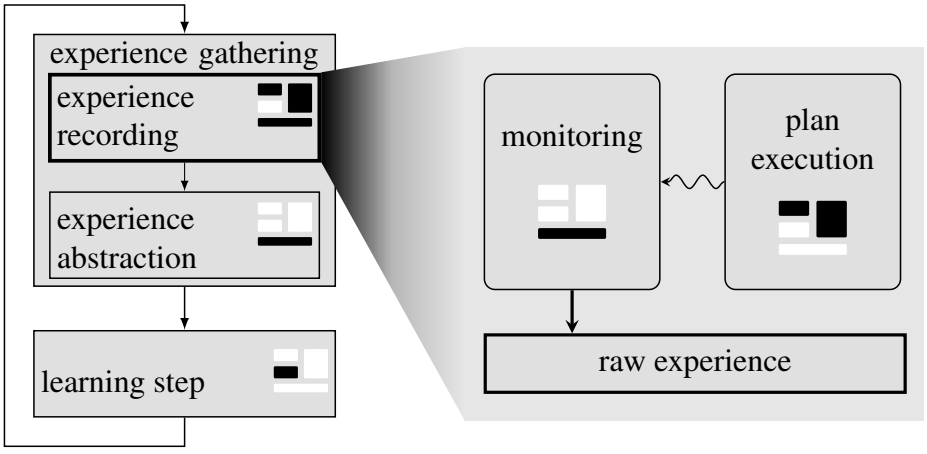
We have now given an overview what experiences are, what subtypes there are and how they are processed. In the following we inspect the steps involved in experience acquisition and processing in more detail and describe the language constructs specific for these steps.

5.2.2 Raw Experience Detection

Raw experiences are the direct observations of beliefs and internal robot parameters during execution. Normally, they are represented as transient experiences in a data structure inside the program that is passed to the next abstraction step immediately.

Figure 5.8 shows the processes involved in raw experience acquisition. The robot is controlled by the performance element, possibly guided by advice from the problem

Figure 5.8 More detailed view on the experience gathering step of the learning cycle. The experience gathering is split in an experience recording and and experience abstraction step. For recording experiences two independent processes are active: one for controlling the robot and one for monitoring.



generator. An independent monitoring process observes internal and external parameters that are changed by the control program. It records relevant data and stores it into the experience data structure. The controlling and monitoring processes operate without direct process communication. The monitoring process starts in a sleeping state and is activated by certain events given in the raw experience specification.

In the following, we explain the relevant input for specifying raw experiences completely and declaratively. First we concentrate on the specifications necessary for defining the observational aspects of the experience, i.e. we disregard the activity of the performance element, which is treated in the next section.

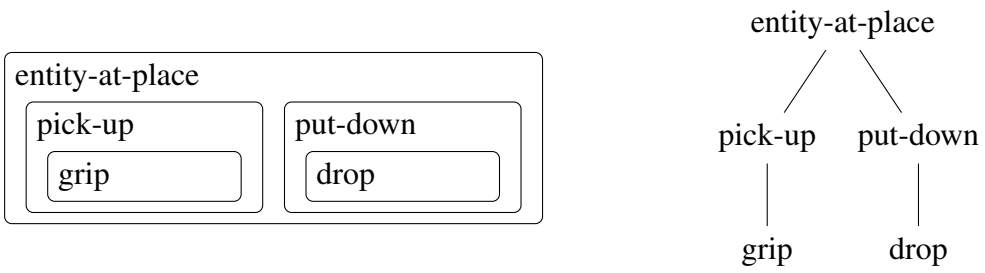
Experience Automaton

We represent an experience by a combination of an automaton defining an episode and the data associated with it. The automaton for raw experiences is defined along two dimensions: the hierarchical structure and its correlation to the program being executed.

The automaton structure itself is described by a hierarchy of subautomata. The mode transitions are not modeled explicitly. The reason is that the transitions happen when the program is executed. The additional specification of transitions might pose another constraint of when the automaton corresponds to the program parts to be observed. We didn't consider this extra complexity to be necessary, because (1) usually subprograms operate in the same way every time they are active, i.e., they call their subroutines in the same order; (2) the additional constraints on execution order can be modeled by events and the handling of the experience data when it is complete (see page 78); and (3) the syntax for specifying experiences would have become more complex.

Figure 5.9 shows the hierarchical definition of an experience automaton — the picture of an automaton on the left and the illustration of the hierarchy of automata on the right. The automaton with the name *entity-at-place* has two subautomata: *pick-up* and *put-down*. The order in which these automata are expected to be activated is not specified. Each subautomaton contains one other mode.

Figure 5.9 Hierarchy of a raw experience automaton.



In this example we see that automata are identified by unique names. In simple cases where only one automaton without submodes is needed, it can be declared anonymous.

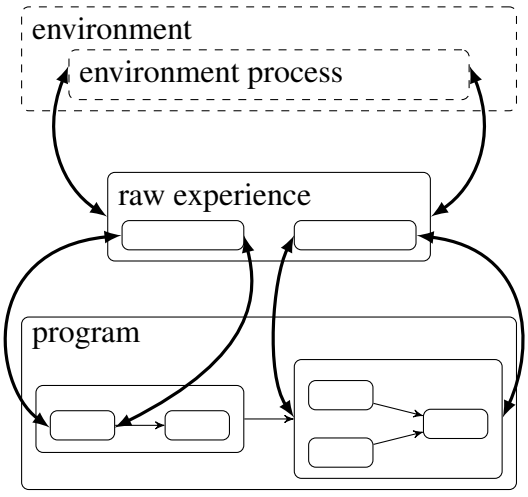
Anchoring

One of the main challenges of experience acquisition is the detection of episodes — interesting time periods, in which to record data. From each episode for a raw experience class, one experience instance can be obtained. In RoLL episodes are defined by anchoring the experience automaton to the control program execution, so that the automaton activation is connected to events during the run of the program.

There are two sources of anchoring as illustrated by Figure 5.10: events in the environment and events inside the program. We have shown in Section 4.3 how the control program can be modeled in the framework of hybrid automata. This means that we only have to associate a subautomaton of the program with the specified experience automaton. To do this, we must access the RPL task network and navigate through it to find the desired automaton represented in it.

Possible introductory points to the task network supported by RoLL are the activation of a (BDI) goal to be achieved, the execution of a routine, or the execution of a globally tagged task. Global tags are an addition to pure RPL that are activated independent of the task context. For example, if a normal RPL tag is positioned inside a `loop` construct, it only comprises the first iteration of the tagged code. Other iterations of the same code are represented by other instances in the RPL task tree and are therefore addressed differently.

Figure 5.10 Description of the HHA structure for raw experience acquisition. The arrows indicate how the experience automaton can be anchored in the automaton defined by the robot program and in the environment process.



In contrast, a global tag is activated every time the piece of code is executed.

After addressing an RPL task via the goal, routine or a global tag, other program parts are accessible by navigating through the task tree with the `sub` expression, which is given a path description and a task. The path description is defined in RPL, see Appendix A and Figure 3.4 on page 33 for more detail.

The anchoring of experience automata in the program structure is very important for learning models of the robot activities. It allows to specify experience that is recorded while the robot is performing a navigation task, for instance. We can also capture more subtle details of the execution of a routine by specifying its subautomata and record the interactions between them.

Other experiences rely more on the state of the environment, for example when learning models of user preferences. In this case, the experience automaton should be described in terms of state variables, which are the internal representation of the environment in the program. In the light of hybrid automata, we represent "environment automata" by giving an invariant. Thus, the environment activity can also be described as an automaton whose activity lasts as long as the invariant stays intact. For example, we might be interested in the automaton that is active while a human is in the room. This same automaton could be described by the two events of a human entering and leaving the kitchen, but in our framework we specify the invariant condition as someone being inside the kitchen.

To summarize, the hierarchical structure of an experience automaton can be anchored to the control program in two ways: (1) by connecting it to the program structure, which corresponds to matching the automaton structure to parts of the program and (2) by associating it with environmental conditions, which corresponds to matching it with an imaginary environment automaton. Both the environment and program automaton don't exist completely, or they exist in different forms. According to the needs at hand, both kinds of automata can be modeled to an arbitrary level of detail. Of course, both methods can be combined so that for instance experiences can be recorded for navigation tasks where people are present in the room.

Experience Data

We have shown how the automaton structure is defined and how it is associated with the imaginary program and environment automata. In the light of Figure 4.8 on page 56, this limits the experience on the timeline, focusing the robot's attention to episodes. Now we have to reduce the theoretical amount of available data during this time by restricting the variables that are to be recorded.

We have seen in Figure 5.5 on page 69 that variable values can either be stored once at the beginning or end of the automaton execution or continuously during its activation. For the events and the interval there can be different sets of variables to be recorded. In the figure, the robot's velocity is recorded continuously, whereas its pose is only saved at

the beginning and end of the automaton activity.

The values to be recorded can stem from global variables like state variables, which are accessible from all over the program. Another source of data are local variables that are changed during program execution, for example the path a robot has chosen for navigation in the current situation. This is possible, because the local variables are accessible via the RPL task tree. The addressing of local variables therefore involves navigating through the task tree like in the specification of anchoring the automaton to the program.

Together with an arbitrary hierarchical structure, very refined experiences can be defined easily. For example, we want to capture the behavior of a routine for picking up objects. We might be interested in the robot's arm trajectory and the time it needs to fulfill its task. This can be defined only by looking at the top-level automaton for the pick-up routine. Besides, an interesting information would be the number of trials in grasping the object or how often and under which conditions it was lost in the attempt of lifting it. For this data, the representation of the automaton must be refined further so that the lower-level grasping routine is represented as well.

Stability with Respect to Failures and Unexpected Events

By giving the structure and data of the experience automaton, the raw experience is completely specified and can be acquired automatically. However, in real-world robotics things often don't go as planned. For one thing, there is the RPL failure system. If failures occur during the execution of the observed program part, the recording is stopped or if the failure is repaired, its occurrence is added as a temporary information. RoLL permits to add specifications of what to do if such an event has been encountered.

The straightforward reactions are to discard the experience or to store it in spite of the failure. But more refined behaviors are possible in RoLL. Before storing it, the data can be modified, added to or be deleted partially. So when a failure has occurred during execution, but has been repaired, the time recorded for completing the task is certainly not correct. The safest reaction would be not to store it permanently. If experiences are scarce in the domain, it might however be advisable to adjust the recorded time according to some heuristics and store the cleaned data. Or, one could discard the time as invalid and only record the other data, which might still be significant.

Beside program failures, there are other events that can be disturbing to experience execution, but pose no problem in the normal run of the program, for example if the dog is jumping around the robot. This class of events could in theory be handled by environment invariants, so that only experiences are recorded when there is no dog in the kitchen. However, parts of these experiences might be useful in spite of the disturbance and can thus be used or modified before saving. So in every automaton, events can be defined by fluents. If the fluent becomes true, the recording mechanism notes that this type of event has happened. After the experience is complete the decision what to do with it is made in

the same way as with failures. Depending on the event, the data can be stored, possibly in a modified form, or discarded.

Finally, we shouldn't count on the recording mechanism to work 100 percent reliably. The update of fluents within the program can sometimes be delayed, maybe because the computer is busy with other activities that have nothing to do with the program execution. Or some state variable changes might not be detected by the state estimation process. In these cases, the stopping criteria of an automaton might be overlooked. Then the automaton activation would only stop when the automaton is activated and terminated the next time, which of course makes the data unusable. A straightforward way to solving this problem is to introduce a timeout restricting the time of automaton activity. If it takes longer, the recording is stopped and the timeout event is notified, so that an appropriate reaction what to do with the data can be defined as in the other cases.

Beside the monitoring of events, RoLL offers another mechanism for preventing the recording of unusable data. Instead of defining all the events that could hinder the observation of experiences, one could have a look at the recorded data itself at the end of the episode. RoLL allows to test if certain data slots in the experience automaton have been filled with data. If some value is missing, the same reactions as for events are available, including the possibility to complete the missing value by heuristic calculations from other recorded values.

At the moment, we are only describing passive experience acquisition, where the robot control process works completely independent from the experience recording process. When thinking of active experience acquisition, the two modules should work closer together. In the light of failure and event handling, the critic element should inform the performance element that the experience was not completely useful, so that the problem generator might consider this information and for example pose the same problem again. In the current version of RoLL this mechanism is not implemented, because the problem generation in general has been treated only roughly. A better support of active experience acquisition is subject to future work.

RoLL Syntax for Raw Experiences

We have now described all the necessary information for specifying experiences. The complete syntax of the `define-raw-experience` construct can be found in Appendix C. Here we only show two examples of how the specification of raw experiences looks like in RoLL and review the points just described.

The first example is presented in Listing 5.1. The experience we define here has the name `navigation-time-exp` (line 1). It consists of only one automaton. Instead of giving it a name, we provide the keyword `:anonymous`. Any keyword symbol can be given for an anonymous automaton, whereas symbols in other packages are interpreted as automaton names. The automaton corresponds to the program automaton for the routine

`b21-go2pose-pid-player` (line 4). When the automaton is activated, the current time, the robot's pose (an object consisting of its x- and y-positions and the orientation) and the goal pose demanded by the goal are recorded (lines 5–8). This latter pose is not stored in a global variable, but is a local variable of the routine. It can be accessed by the `:internal-value` keyword (line 8). For getting the desired pose the functions `goal` and `pose` must be applied to the internal value. When the automaton stops, the time of this event is also recorded. If the execution of the automaton is either aborted by some more urgent process or fails, the timestamp at the end of the activity is set to 0 and the information is stored (lines 10–12).

Listing 5.2 shows a more complex example for observing the activity of gripping objects. In this case we have omitted the data part (the *begin*, *end*, and *interval* parts) of the experience and only show the structure, anchoring and event handling. The hierarchy of automata is the same as in Figure 5.9 on page 75. The outermost automaton is defined by the goal `entity-at-place` (line 4). The subautomata are all defined in terms of the task path from the parent process to the subprocess. The keyword `:parent` is an abbreviation for giving the name of the parent automaton. Because in this example several automata are involved, they must all be identified by a unique name. The experience handling is more complex here. It is formulated in the form of a LISP `cond` expression. The first condition to be tested (line 21) is if the *fail* event has occurred in any automaton during execution. If this has happened, the information is discarded. The next condition (line 23) tests if the abort event has been noted for the *put-down* automaton. If the *put-down* automaton has been aborted, some data is modified (indicated by dots) and stored. Another possibility for deciding what to do with the recorded data is illustrated in the third condition (line 26). It tests if all the desired data at the end event of the main automaton has been recorded. If this is not the case, some data is modified and then stored. The default case — if nothing has failed, the *put-down* automaton has not been aborted and the data at the end of the

Listing 5.1 Definition of a raw experience for recording the time of a navigation task.

```
1  (roll:define-raw-experience navigation-time-exp
2   :specification
3    (:anonymous
4     :rpl (:routine-execution 'b21-go2pose-pid-player)
5     :begin ((timestep [getgv 'statevar 'time-step])
6             (start-pose [getgv 'statevar 'b21-pose])
7             (goal-pose
8              (pose (goal (:internal-value "ROUTINE" :this :exact)))))
9     :end ((timestep [getgv 'statevar 'time-step])))
10  :experience-handling (((or (:event :abort) (:event :fail))
11                         (roll:replace-data (:var timestep :end) 0)
12                         :store)))
```

main automaton has been recorded — is to store all the data.

In this section we have shown how raw experiences are defined in RoLL, namely by giving the automaton structure, its anchoring in the control program and the desired data. To make the collection more reliable, several mechanisms for manipulating data before it is stored are implemented. With the declarative specification they can be recorded automatically using the construct `acquire-experiences`, which takes an experience data structure as an argument and creates a process that runs in parallel to the control program. After one experience has been recorded, it is immediately processed to a more abstract one.

Listing 5.2 More complex example for observing the activity of gripping objects.

```
(roll:define-raw-experience used-arm-cup-exp
 :specification
  (main
   :rpl (:goal-subtask (:goal-execution 'entity-at-place))
   :children
    ((pick-up
      :rpl (:achieve-subtask (:sub ((tagged pick-up) (try 0) (step 2)
                                    (process-body) (step 1))
                                   :parent))
      :children ((grip
                  :rpl (:sub ((tagged grip)) :parent))))
     (put-down
      :rpl (:achieve-subtask (:sub ((tagged put-down) (try 0)
                                    (step 2) (process-body) (step 1))
                                   :parent))
      :children
       ((drop
         :rpl (:sub ((tagged drop) (try 0) (step 2)
                     (process-body) (step 1))
                    :parent))))))
  :experience-handling (( (:event :fail)
                          :discard )
                        ( (:event (:abort put-down))
                          ...
                          :store )
                        ( (not (:available (:end main)))
                          ...
                          :store )))
```

5.2.3 Problem Generation

The difference between active and passive experience acquisition lies in the way the performance element works during experience recording. In the former case the robot is controlled in a way such that the actions will provide good learning experience. When experience is acquired passively, the actions are selected according to the agent's normal utility criteria regardless of their use in observing experiences. Put another way, active experience acquisition needs the problem generator of the agent architecture depicted in Figure 2.1 on page 17, whereas passive experience gathering does without.

The main focus of this work is learning with passive experience acquisition. It seems natural that an agent should observe its activities while performing them, because humans usually learn in the same way. Besides, learning and the recording of experiences in particular, are executed without degrading the robot's normal activities. On the other hand, active experience acquisition provides experience data that is tailored better to the learning problem. The needed data can be gathered faster and less data is required, because redundant experiences can be avoided. We can assume a household robot to have idle times, so that active experience acquisition could be performed primarily in these time periods. In this way, the overall robot performance would not decrease.

For gathering experiences actively, a special control program containing a set of free parameters is executed. By varying the parameters different experiences can be observed. The relation between program parameters and observed experiences is non-trivial and usually non-deterministic. Therefore it is difficult to determine good program parameters a priori. This means that the problem generator should create parameterizations at run time depending on the experiences already acquired. Information about the already existing experiences can be got from the critic element, where the experiences are stored and managed. The main challenge here is to define the set of needed experiences for learning on an abstract level, determine if the available set of experiences fits this specification, and what must be done to acquire missing experiences. To our knowledge this topic of defining experiences and establishing connections between the program executed by the performance element and the observed experiences has not been subject to any research and is certainly not straightforward.

Other interesting questions arise when active and passive experience acquisition are combined, a topic that has received much attention, especially in the context of reinforcement learning (Thrun 1998a; Cohn 1996; Yu, Bi, and Tresp 2006). The agent has to find an appropriate trade-off between exploration (active experience acquisition) and exploitation (passive experience acquisition) when it has to learn, but also execute its primary tasks. If exploration is performed excessively, only the learning component benefits from the robot's actions, whereas the main task is achieved suboptimally, if it is achieved at all. On the other hand, pure exploitation may lead to poor learning results and prevents the observation of certain parts of the state space.

Beside the challenges posed by active experience acquisition, it can, on the other hand, make the observation of experience data easier. When recording data passively, all information about the primary process executed by the performance element must be got from the internal program structure and changes in state variables. When acquiring experience actively, the main program is written only for the purpose of observing experiences, so that communication between the execution process and the monitoring can provide extra information. This means that the primary process can tell the critic element explicitly when it should start and stop recording data. Furthermore, failures or other unexpected events can be registered by the observing process and be reported to the executing process, so that problems can be repeated if no meaningful data could be extracted from their first execution.

Overall, there is a lot to be said and done about active exploration. RoLL is mainly designed for passive experience gathering, but provides basic facilities for active acquisition, which correspond to the state of the art for pure active experience acquisition and makes available the means for implementing more sophisticated modes of problem generation. It is possible to implement dedicated control programs for experience acquisition and define parameterizations for them a priori. In terms of process communication, interesting program parts can be registered by a global tag, so that their execution can easily be detected by the monitoring process. Combinations of active and passive acquisition are not provided by RoLL explicitly, but can be implemented for specific problems.

Problem Generator

For exploring the world actively, we assume that the performance element runs a given program subsequently with different parameterizations. For example, when we want to learn the time prediction model of the robot's navigation routine, we might run a program that navigates the robot to different places in the world. The task where to go is specified by a vector containing the x- and y-coordinates of the goal position and possibly a goal orientation, constituting the free parameters to be varied. The navigation program is then executed several times with different target locations.

Thus, the complete experience-gathering plan is given by a program specification and a list of parameters. The simple problem generator implemented in RoLL can produce parameter vectors before the execution starts. In the following we introduce a simple language allowing the specification of such vectors.

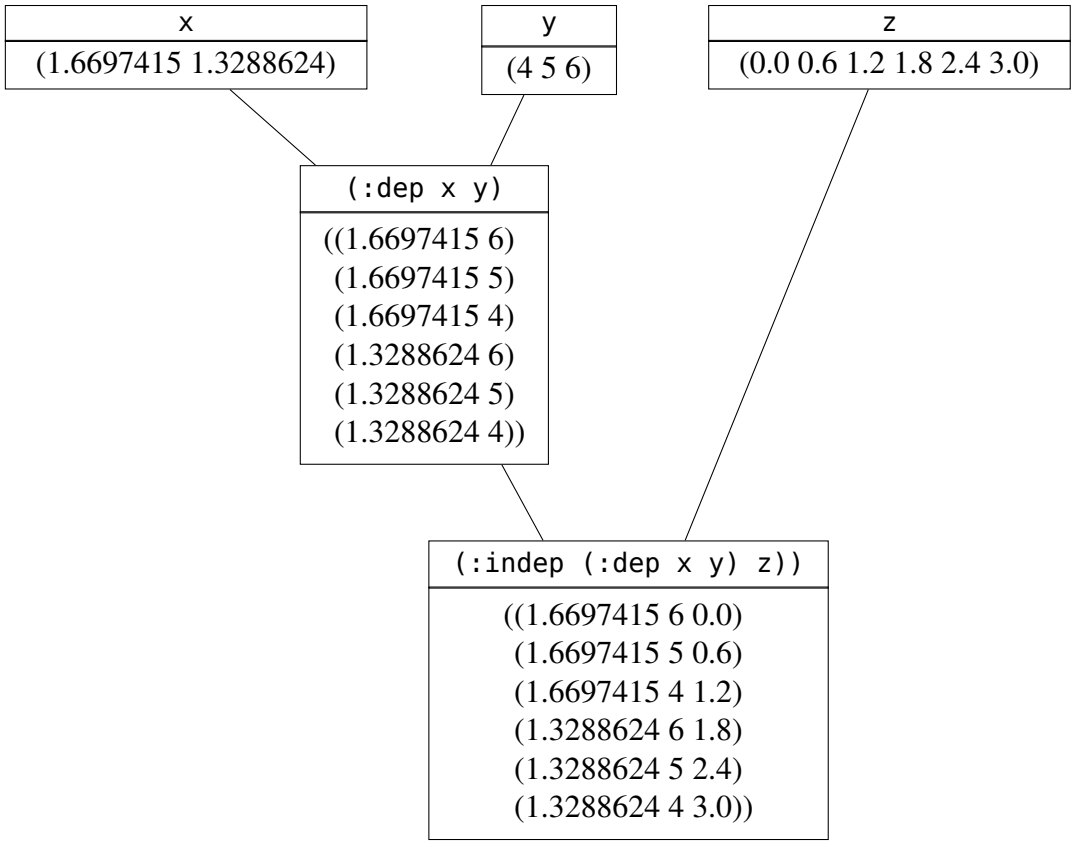
First of all, for each variable we have to specify how many and which values are to be tested. One possibility is to generate values randomly, only giving the possible range of values. In the navigation example this would correspond to sending the robot to uniformly distributed positions in the world. Another option is to specify fixed values by hand, for example only allowing angles of 35, 123 and 260 degrees for the goal positions. Thirdly, the state space can be discretized and be tested completely within this simplified

representation. In the kitchen, for instance, we could define the limits of the kitchen and use only the positions every 0.5 m. Then the problem generator would generate all positions on the specified grid.

The possibilities for specifying values apply to each variable separately. This means we can define goal positions by placing a grid on the coordinates in x-direction, determine y-coordinates randomly and predefine some values for the target angle. So for every parameter variable the problem generator produces a list of values to be used in the performance element. The lists of the single variables must then be combined into a list of vectors containing a parameterization for each variable. If two variables depend upon

Figure 5.11 Working of the problem generator in RoLL illustrated by an example.

```
(problem-parameters
 :parameters ((x :random :min 1.0 :max 2.0 :samples 2)
              (y :predefined '(4 5 6))
              (z :cover :min 0.0 :max 3.0 :interval 0.6))
 :relation (:indep (:dep x y) z))
```



each other, the single lists are combined by a cross product operation. In contrast, if they are independent, the lists are combined item-wise (see examples below). The dependency relations of variables can be nested and combined arbitrarily.

By generating values for a variable, sometimes the number of generated problems is fixed. This is the case for a list of predefined values and for a complete coverage of a discretized state representation. For random values there are no restrictions on the number of samples generated. In some cases, the number of randomly generated values can be determined by the combination mode of different variables. If two variables are independent, the number of samples for one of them is fixed and the other one is generated randomly, then the overall number of problems is equal to the length of the value list of the first variable. If the number of desired random values cannot be determined automatically, it must be specified explicitly.

Ambiguities can arise when two lists of values with a predetermined length are to be combined as independent variables, if the lengths of both lists are not equal. One option is to use the smaller number of values and discard the superfluous ones, the other one to use the higher number and repeat values from the shorter list.

Figure 5.11 demonstrates how the declarative specification of the problems is transformed to a list of parameters. In the first step, the value lists of the specified parameters are generated. These lists are then combined step by step according to the dependency relation. More examples can be found in Section C.3.3.

Overall, the problem generator in RoLL proposes a powerful language for specifying parameter settings for different runs of experience acquisition. Although its functionality doesn't comprise all the possible features of problem generation, it provides the means of performing active experience acquisition in a way it is done in most state-of-the-art robotic systems.

Process Communication

When acquiring experience actively, explicit communication between the performance element and the critic element could be established for facilitating the detection of episodes and reacting to failures or unexpected events. RoLL doesn't provide explicit means for this kind of communication, but it does include means to include them for specific problems.

The detection of episodes can be facilitated by setting a global tag around the code that is to be detected as an episode. Then it isn't necessary to navigate through the task tree and the interesting time period of the execution is identified unambiguously.

If there are events that can easily be detected by the execution module, but are difficult to register by the monitoring module, a global fluent can be introduced that is changed by the execution module and whose values are observed by the monitoring module. Events detected that way can be used for deciding what to do with recorded data after an episode

has ended.

Communication from the monitoring process to the execution module is not possible directly. However, events detected by the critic element can be coded into the acquired experience. The execution module has access to all recorded experiences and by examining if an experience has been recorded and what data it contains it can conclude if there were any problems detected during the recording in the last episode. If some unwanted effect has occurred, the episode can be repeated.

Overall, although there is no explicit communication between the performance and critic elements, RoLL provides means for interchanging information between the two modules. As active experience acquisition is usually not too complex, the standard functionality of RoLL suffices in most cases. For special problems, there are ways to establish communication between the observing and executing processes.

Extendability to More Sophisticated Problem Generation

At the beginning of this section about problem generation we have explained what challenges are posed on sophisticated experience acquisition. Although it is difficult to find a general solution and good representation for the desired set of experiences, solutions for specific problems can be found and implemented with RoLL.

We have explained that it is extremely difficult to represent the desired experiences in a general way and ensure that the wanted distribution of data is acquired by a dexterous coordination of the execution and monitoring modules. This involves a thorough understanding of the underlying state space of the problem and knowledge of how the robot's activities relate to the acquired experiences. Then the execution module could check the available experiences and notice what parts of the state space must still be explored, whereupon it selects appropriate problems to execute. After that it checks again if the desired experiences have been observed.

We have also mentioned the possibility to mix active and passive experience acquisition. This can be simulated in RoLL, too. The control program can be written in a way that the state space is explored actively first and later a gradual transition to passive experience acquisition takes place. The drawback of this approach is that the code for active acquisition is never removed from the program and adds complexity to it that is not necessary after the learning process has ended or switched completely to passive observation. On the other hand, different exploration strategies can enhance learning and RoLL makes it possible to test different strategies against each other.

Without providing explicit means for active experience acquisition that goes beyond the methods used in current systems, RoLL enables the implementation of additional features for specific problems. This can be the starting point for a more thorough investigation of the features needed for active experience acquisition and representations of the experiences needed for learning.

5.2.4 Experience Abstraction

Up to now we have only described the first step in getting experiences for learning, that is to say, the gathering of the raw experiences. Usually unsuitable for learning, these experiences must be converted to a more abstract form. This process can be performed in several steps or by only one conversion, depending on the learning system, the problem and interdependencies between learning problems.

In the following we first present an example of how experiences are abstracted for giving an intuition of how the abstraction process takes place and what challenges are connected with it. After that we provide an overview of the demands and design criteria of the language constructs involved in experience conversion. Finally, we explain the operations on experiences that lead to more abstract representations and how they are implemented in RoLL.

Example

As an example we have a look at the learning problem of a function deciding which of its two grippers a kitchen robot should use for grasping an object. Figure 5.12 shows the steps involved in the abstraction process. Every experience consists of a hybrid automaton and data. The automaton structure is depicted on the left for each abstraction step and the data associated with it on the right. The automata are depicted according to the representation in Figures 4.4 on page 49 and 5.5 on page 69.

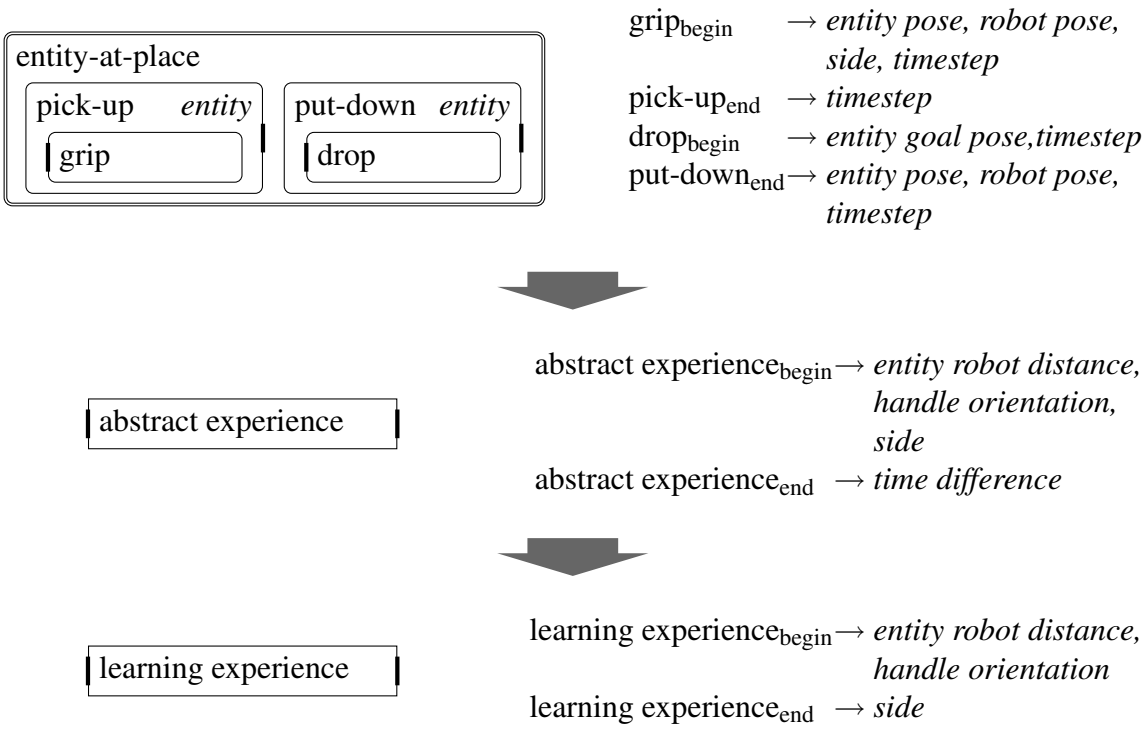
On page 81 we have already presented the automaton for the raw experiences of this problem. It comprises a hierarchy of automata each corresponding to a part of the control program. The outer automaton is linked to the goal `entity-at-place`. When this goal is to be achieved, two routines are called sequentially (although the order is not included in the representation of experiences!): `pick-up` and `put-down`. The `pick-up` routine comprises the whole process of moving to the object, grasping and lifting it. The grasping procedure as such is performed by the routine `grip`. Similarly, the pure action of releasing the object is performed by the routine `drop`.

In our problem we are interested in the time from when the robot starts to touch the object for gripping until it has lifted it to the carrying position and the period between moving the object to the final position and removing the arm. Therefore, we need data at the beginning of the `grip` and `drop` routines and when the `pick-up` and `put-down` routines are finished. The data we need comprises the initial position of the entity to be carried, its desired goal position and the place where it is actually placed by the robot. Besides, the time needed for performing the lifting and dropping processes is important for comparing the performance of the robot hands. Finally, we must record which hand is used in the gripping process. Our robot cannot hand over objects between gripping and releasing, so the side chosen for gripping is the same as for putting down the object.

Figure 5.12 shows how the raw experience must be defined for getting all this information. When the gripping process starts for fulfilling the `entity-at-place` goal, the current positions of the object and the robot, the hand that is used (left or right) and the time when the action starts are recorded. The time of when the routine `pick-up` is finished is also important for determining the duration of the whole gripping action. Similarly, the duration of the dropping process, the intended position of the object, the actually reached object position and the robot's position while setting down the object are logged.

Now we can convert the raw experiences into abstract ones. The automata for the abstract experiences are depicted as a function automaton. In fact, the automaton type is irrelevant for RoLL. In the illustration we wanted to underline the different automata for raw and abstract experiences. Whereas the automaton structure of the raw experience corresponds directly to the processes of the program, the structure of the abstract experience can best be seen as a function mapping the initial situation and the arm used to the time needed for gripping an object in the described situation with the specified hand[1]. The de-

Figure 5.12 Experience abstraction for the learning problem of determining which hand to use for gripping.



[1] Abstract experiences in general are not restricted to this simple automaton type. They can be nested analogously to raw experience automata.

scription of the initial situation is reduced to the distance between the robot and the entity and the relative orientation of the object with respect to the robot. It is noteworthy that the abstract experience doesn't make use of all the data contained in the raw experience. This can happen when raw experiences are used for different abstract experiences or when it is not yet clear which data is needed for the best learning results. In the example presented here we noticed that using the information of putting the object down enhances the learning result significantly (see Chapter 6 for details). For keeping the example simple, we present here the version where we used only the information of the gripping process.

The experience obtained by the first abstraction step could be used for learning a model of the gripping routine — how long does it take the robot to grip an object in the specified situation with a certain gripper? If we want a decision function to choose the gripper, we should go one step further and produce an abstract experience where the situation of the robot and the object is mapped directly to the gripper to be used. For this, it has to be decided for each gripping situation, which side is the best according to the time that was needed. This means that compared to the former abstraction step, the learning experience contains only half of the data, the disadvantageous examples having been discarded.

As we have seen, experiences are always represented as an automaton and data corresponding to it. RoLL provides constructs for defining the experience types and for translating one experience type into another.

Design Criteria

The main problem in designing language constructs for experience abstraction is the great variety of possible demands. As they depend on the learning problem and all possible cases can hardly be foreseen, the language should be general enough to allow all algebraic and conditional translations between experience data.

Furthermore, we have described the different classes of experiences, transient and persistent, which can be saved in a variety of ways, like in a database or a log file. The conversions between them must be specified in a declarative manner for being independent of the underlying experience class.

Besides, the experience conversion should fit into the context of the whole language. The syntax should be logical and comprehensible and as similar to the definition of raw experiences as possible, while considering the peculiarities of abstract experiences. Therefore, the natural choice for representing abstract experiences is within the framework of hybrid automata.

Although it is possible to abstract experiences in several steps, the choice of how many steps are needed should be determined by the programmer and not be restricted by the language. Therefore, it must be possible to describe complex transformations, so that no superfluous transformation steps must be added.

Operations on Experiences

Up to now we have used the words “abstraction”, “conversion” and “transformation” synonymously. However, conversion or transformation can be more than just pure abstraction. Experiences can be converted along two dimensions. It is possible to take one experience instance and transform it to another experience instance of a different type. This is what we call “abstraction” in the narrower sense. Another possibility is to map a set of experience instances to another set of experience instances of the same or another experience class. We call this process “filtering” of experiences. It can for instance be used to reduce several original experiences to only one result experience, for example by calculating the average of several values. Notwithstanding the name, the result set of a filtering operation can be larger than the initial set of experiences.

Put in a simplified way, abstraction primarily works more on the automaton structure, whereas filtering affects the data traces. The operations on automata were defined in Section 4.4.2 as abstract, expand and restrict operations, which rely on the hierarchical structure of the hybrid automata. In RoLL, the distinction of these operations is not made explicit, primarily for not restricting transformations in any way and for making factitious conversion steps unnecessary.

The filtering operation is only supported in a rudimentary fashion by RoLL. Only the operation of discarding an experience instance is implemented explicitly. This is due to the fact that filtering relies strongly on the underlying experience storage system and on the problem at hand. With knowledge of the used experience class it is however possible to add filtering mechanisms to given problems.

For experience conversion in RoLL, first of all the structure of the resulting experience must be defined very similar to the way raw experiences are specified. After that a con-

Listing 5.3 Abstract experience definition, where experience and conversion are combined in one specification.

```
1  (define-abstract-experience abstraction-stage-1
2   :parent-experience raw-experience-for-gripping
3   :specification
4    (:function
5     :begin ((entity-robot-distance
6              (distance (:var entity-pose (:begin grip))
7                        (:var robot-pose (:begin grip))))
8             (handle-orientation
9              (relative-orientation (:var entity-pose (:begin grip))
10                                   (:var robot-pose (:begin grip))))
11            (side (:var side (:begin grip))))
12     :end ((time-difference (- (:var timestep (:end pick-up))
13                               (:var timestep (:begin grip)))))))
```

nection between experiences is established by defining conversions between experiences, always in the direction from raw to more abstract experiences. On page 71 we mentioned the possibility to get instances of an abstract experience type from several types of raw experiences or to use a raw experience for generating different kinds of abstract ones. This is the reason why the definition of the experience structure and the conversion are different concepts. However, in most problems there is a one-to-one relation between raw and abstract experience. Therefore, RoLL offers a simpler method for specifying the experience structure and the conversion in one definition.

Listing 5.3 shows a simple example of an abstract experience definition, where we specify the experience structure and the conversion in one definition. Therefore, after naming the experience `abstraction-stage-1`, the first parameter gives the raw experience type serving as the starting point for the conversion (line 2). Then follows an automaton specification that looks almost identical to automaton definitions for raw experiences. Only the source of the data is given in a different manner. Here the values are not observed, but calculated from values of the raw experiences. These values are addressed by the special syntax explained in Section 5.2.1 on page 70. For example `(:var entity-pose (:begin grip))` addresses the value of the parameter `entity-pose` that was recorded at the activation of the grip automaton. These values can be combined by any LISP expression, for example by the function `distance` (lines 6,7 in the example), which calculates the Euclidean distance of two points in 3D space. For a complete overview of the syntax for abstract experience specifications see Appendix C.

Experience Processing in Different Experience Classes

We have presented the declarative part of experience conversion. Now we explain in more detail how the definitions are executed taking into consideration the need of abstracting away from different storage systems and the differentiation of permanent and transient experiences. Including different types of storage systems into RoLL can be done by defining an experience class, which takes care of the interface between the external storage and the experience processing in RoLL.

The conversion process between experiences of different levels of abstraction is depicted in Figure 5.13. Each experience, no matter if transient or permanent, is represented by one reference instance of the according LISP class[2] (in the figure depicted as data boxes called E1, E2, E3, and E4).

The abstraction process is divided into two operations to be supported by all storage

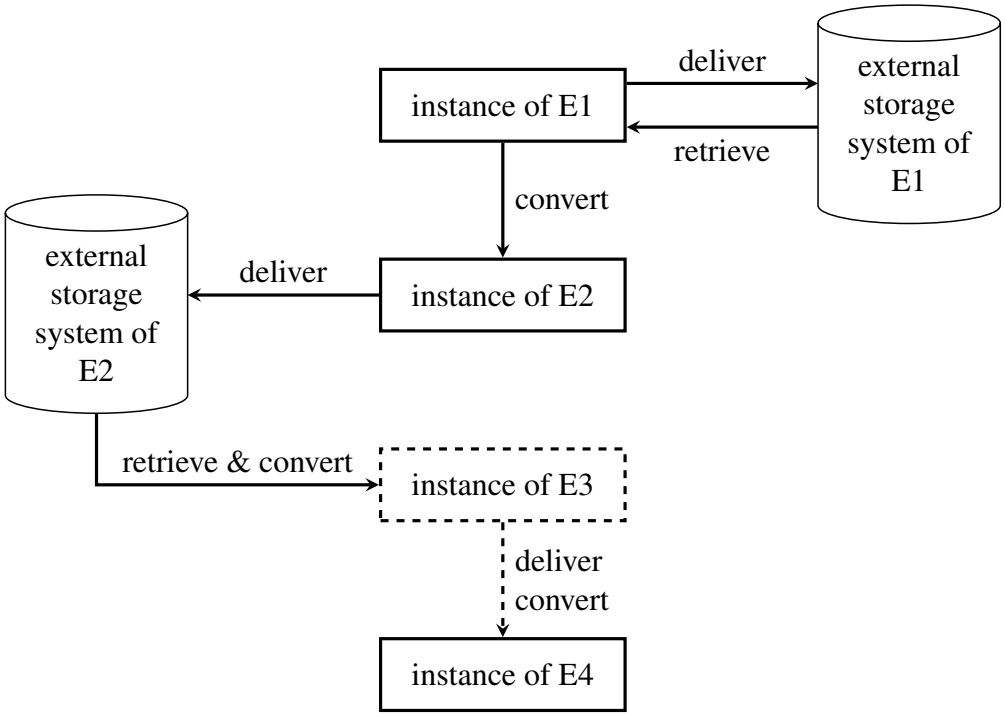
[2]The notion of experience classes and instances is a little tricky. When an experience is specified a LISP class is generated and the episode data is stored in instances (or only one reference instance) of this class. The generated classes of experiences representing an episode are derived from more general experience classes defining the storage system to be used. The storage defining classes is what we normally refer to as "experience classes" in this work.

systems: `deliver-experience` and `convert`. First an experience is filled with the data it may contain according to its structure. It is then informed that the information is complete by the call of `deliver-experience`. The experience then has to store the experience away in order to be open for new experience data. The storage can involve external storage systems like databases or files as in the case of E1 and E2 in the example. E3 — a transient experience — doesn't really store the data, but passes it on to the next level of abstraction.

The second method every experience class must provide is the method `convert`, which receives the instances of the two experiences to be converted, e.g. E3 and E4, and destructively replaces the values stored in the instance of E4 by the values computed from the data in the instance of E3 and the abstraction specification between E3 and E4. For persistent experiences, the conversion is regarded as a two-step process: retrieving the data from the external storage system and abstracting it according to the given specification. The experience class E1 implements this procedure in a straightforward way: it retrieves the data from the external storage medium and stores it in its reference instance. The conversion method applied between E1 and E2 is the same as if E1 was a transient experience.

Some storage systems provide more functionality than just saving the data. For example, when using a relational database sophisticated calculations can be performed in the query. Instead of first reading the data as it is and then converting it by means of LISP,

Figure 5.13 Conversion process for different experience classes.



one can do the whole abstraction or only parts of the calculations involved in the external process. This procedure is shown in Figure 5.13 between experience classes E2 and E3. In some cases, the external system can perform operations more optimal (e.g. cross product on tables in a database) and the calculations might reduce the data passed between the external and the LISP process.

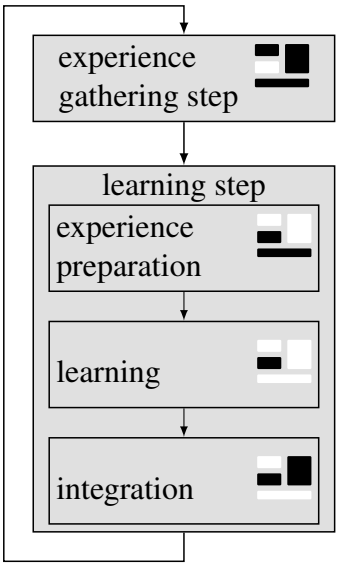
The operations `deliver` and `convert` are the minimum operations required for an experience class to support. As experience conversion only considers one experience instance at a time, it naturally restricts the possibility to compare experiences to others from the same class. However, if the storage system gives access to all experiences and allows operations on them, this functionality can be added as an additional operation on this class of experiences. For example, when using a database, experiences can be filtered and smoothed by data mining mechanisms.

To summarize, the RoLL operations connected with experiences comprise the observation of raw experiences, the generation of action sequences to observe useful experiences, and the abstraction process, which can work with user-defined experience classes.

5.3 Learning Problems

We have described in detail the RoLL language constructs concerned with experiences. Now we explain the learning part. As mentioned in Section 3.1 on page 25, a robot control program includes a variety of function types to be learned, e.g. routine and environment

Figure 5.14 Learning step in the learning cycle.



models, arbitration functions, etc. RoLL offers the specification of different kinds of learning problem classes. This is important for the integration of the learning results into the program. A routine model must be added to the program in a different way than a routine. Secondly, the functionality of RoLL doesn't rely on a specific learning algorithm (or learning system) like a decision tree learner or neural network simulation, but rather offers the possibility to add new ones. Finally we show, how a learning problem is defined in RoLL. The specification of learning problems is declarative, like that of experiences. The declarations are then transformed to executable code automatically.

Figure 5.14 shows the learning part in the context of the whole learning procedure. First of all, the experience, which has already been abstracted, must be adapted to a syntactic form suitable for the learning system. The learning is then performed with the specified learning system, which is usually an external program. Finally, the learned function is integrated into the control program. This is done automatically, depending on the learning system and the learning problem class.

5.3.1 Learning Problem Classes

Robot programs have a variety of functions that can be learned. These include models of the environment and robot behavior, arbitration functions for deciding which routine can achieve a goal, and low-level control routines. The latter aren't actually functions (returning a value), but plans (controlling the robot). But a low-level routine can be implemented as a control loop, the control commands being generated by a function in every iteration.

Beside these standard types of learning problems, the architecture of a specific robot might make other kinds of learning problems necessary, for example functions determining parameters during run-time like the hand to use for gripping or the exact position to put an object at.

Because not all classes of learning problems can be foreseen, RoLL offers a way to add new ones. Learning problem classes differ in their signature, i.e. the parameters they take, and the way the resulting function is integrated into the control program. For example, for learning a model of some control routine, the learning problem must be given the name of the routine and a specification of what kind of model is to be learned (time or failure, for instance). In RoLL, models are tightly coupled to the routine they describe. Therefore, a learned model must be associated with the routine it characterizes. The integration of any routine model is described in the learning problem class, so that for each learning problem of this type the integration can be performed automatically.

5.3.2 Learning Systems

RoLL is not designed for a specific learning paradigm. Any experience-based learning algorithm can be used with RoLL. In fact, the core layer of RoLL doesn't include any

learning system. This means that learning can only take place after at least one learning system has been added. In our experiments we have used two external programs as learning systems: SNNS (Stuttgart Neural Network Simulator) (Zell and others 1998) for neural network learning and WEKA (Witten and Frank 2005) for different kinds of decision tree learning (classical decision trees, regression and model trees).

Each learning system requires the input data to be in a special format. As RoLL allows the definition of different classes of experiences with different kinds of data storage (as described on Page 73), this data format can be implemented as an experience class. The only difference to other experience classes is that there is no need to read from the underlying storage to get the data back into the program, as is required from other experience types. Only the transformation of the generic experience format to the format of the learning system must be specified (i.e. the method `deliver-experience`).

A learning system may demand a special semantic format for the experience automaton. For example, the already integrated learning systems for neural networks and decision trees assume that the experience consists of only one automaton with a begin and end specification. The variables stored in the *begin* part are used as input values, the ones from the *end* part as output values. For other kinds of learning, e.g. unsupervised learning, other data formats can be postulated by the experience for a learning system.

Usually, the conversion from an abstract experience to the experience used by the learning system is not only a shift of data types, but contains changes in data, too. Often, data for use in neural nets should be normalized to a certain range of values, depending on the settings used in the network. In other cases, where no normalization is necessary, the conversion can be a pure transformation from one experience type to another.

Beside the experience type, a learning system in RoLL must provide the call to an external program or the working of a learning algorithm implemented in LISP. Usually, learning algorithms can be adapted to the problem by a set of parameters. The learning system can specify a set of such parameters and use them when calling the learning algorithm.

External learning systems usually provide their results in a form other than a LISP function. The learning system has to take care to convert this format to executable LISP code. This can involve the call of a foreign function, like a C function, or the parsing of a decision tree file, for instance. The embedding of the learned function is specified by the learning system and the learning problem class. The learning system has to provide executable LISP code, whereas the learning problem class defines how this code is integrated into the program.

When a function has been learned, it should be available for use from that time on, even when the system has to be restarted. This means that the result of a learning process should be written to a file that is automatically loaded when the system starts. As RoLL doesn't assume a certain directory structure for projects, the user has to take care that an appropriate path is specified in the learning system.

5.3.3 The Learning Problem

Once the learning problem class and an appropriate learning system have been added to RoLL, learning problems can be defined and executed. Every learning problem can be addressed by a unique name. For performing the learning process, the function `learn` must be called with the learning problem as an input parameter. The `learn` function should only be called after the experiences have been acquired. In the current version of RoLL the scheduling of experience acquisition and learning is performed manually, therefore both processes are treated independently.

For specifying a learning problem, it is necessary to state what is to be learned by giving a learning problem class with appropriate parameters. For example, for learning the time model of the routine `navigate`, the specification according to the learning problem class of routine models is `(:model navigate :time)`. The order in which the parameters `navigate` and `:time` are to be given is not part of RoLL, but is defined by the learning problem class `:model`.

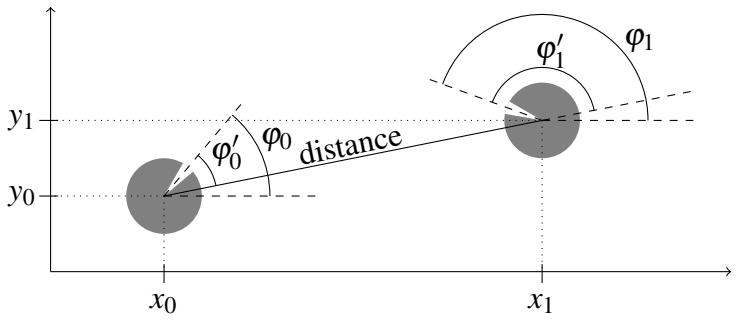
Secondly, the experiences that are to be used for learning must be specified. The experience given in the learning problem must be of a type that is understood by the learning system.

The learning system to be used is the third component of the learning problem specification. It also includes the parameterization of the learning algorithm for the problem at hand.

Finally, the abstracted values used for learning must be associated to the input values of the resulting function. We explain this point in more detail, because it is an important step in the context of experience abstraction.

Consider a low-level navigation routine to be learned. The raw experiences might be gathered by controlling the robot in a random way and recording pairs of start and end points together with the low-level navigation commands, which is a vector of the form $\langle x_0, y_0, \varphi_0, x_1, y_1, \varphi_1, rot_0, trans_0 \rangle$ with the robot's position at time 0, its position at

Figure 5.15 Illustration of experience abstraction for learning a navigation routine.



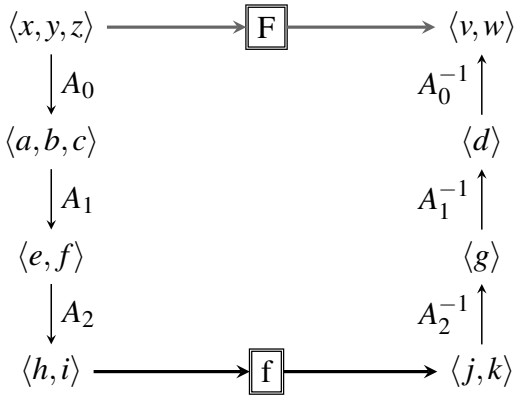
time 1 and the commands given at time 0. After observing such a vector, we can expect the robot to reach position $\langle x_1, y_1, \varphi_1 \rangle$ from position $\langle x_0, y_0, \varphi_0 \rangle$ if it gives the command $\langle rot_0, trans_0 \rangle$.

This experience representation might not be suitable for learning, because absolute positions are given and therefore identical navigation situations are treated as distinct cases if the positions are translated or rotated in 2D space. Therefore, the experiences are abstracted to a form $\langle distance, \varphi_0, \varphi_1, rot_0, trans_0 \rangle$, the correlation to the original experience being depicted in Figure 5.15. Now the signature of the learned function is $distance \times \varphi_0' \times \varphi_1' \rightarrow rot_0 \times trans_0$. However, in a world with many obstacles, the original representation with absolute positions might be more appropriate, since navigation paths must be chosen according to the location of the obstacles. So for learning a navigation routine, different experience abstractions are possible. The call to the learned function should be the same, no matter which abstraction is used for learning, in the example $x_0 \times y_0 \times \varphi_0 \times x_1 \times y_1 \times \varphi_1 \rightarrow rot_0 \times trans_0$. However, the functions produced by the learning system expect to be called according to the abstracted experience.

Figure 5.16 illustrates this phenomenon. The function F is the one that is intended to be learned with the signature $x \times y \times z \rightarrow v \times w$, whereas f is the function produced by the learning algorithm with the signature $h \times i \rightarrow j \times k$. The experience data is prepared for learning by a multi-step abstraction process involving abstractions A_0, A_1 and A_2.

When F is called, it gets the originally intended input values x, y and z, which must be converted with the same abstractions as the learning experience and can then be used to call f. The output of f is not exactly what the caller of F expects. Therefore, the output values $\langle j, k \rangle$ must be transformed to $\langle v, w \rangle$ by applying the abstraction chain A_0, A_1, A_2 backwards. This whole procedure has two tricky parts: (1) How can the original abstraction definitions be used, i.e. how do the values $\langle x, y, z \rangle$ correspond to the values of the raw experience? and (2) How can the reverse abstractions be calculated?

Figure 5.16 Abstraction in the context of the whole learning process.



To illustrate the first question, consider again the example of the navigation routine to be learned. The position $\langle x_1, y_1, \varphi_1 \rangle$ is obtained by random control of the robot. In contrast, for the resulting function, these values come from the goal position, which is given as the input. Thus, for applying the abstractions, which have already been specified, RoLL must be told that the pose $\langle x_1, y_1, \varphi_1 \rangle$ of the raw experience corresponds to the goal position of the function F. The position $\langle x_0, y_0, \varphi_0 \rangle$ needn't be specified further, because in both cases it denotes the robot's current position. Then the abstraction steps for the input values of f are generated automatically.

RoLL also offers the possibility to specify the complete input abstraction explicitly. This means to ignore the already defined abstractions for the experiences and to define a function converting the input of F to the input of f. However, one has to take care that when the experience abstraction is changed, the input abstraction to the learned function must be changed as well. Therefore, the automatic generation is to be preferred.

For the back transformation of output values, RoLL only offers the manual method of specifying a function that converts $\langle j, k \rangle$ to $\langle v, w \rangle$. This is because for the input conversion the abstractions are known, but for the output conversion the reverse abstractions would have to be generated. This might be possible in simple cases, for example for algebraic expressions, but can be really hard when complex LISP functions with conditionals and recursion or loops are involved. Therefore, the reverse abstractions for A_0, A_1, A_2 would have to be specified by hand, which is however more costly than just defining the complete back transformation of the output values.

The specification of the starting point for abstracting the input values and the transformation of the output values of f defines the signature of F. A good signature for F is one that corresponds closely to the input given by the learning problem class. For example, routine models are always called with the routine and the goal it is to achieve. The output should fit the chosen kind of routine, e.g. the time needed for performing the task. The input and output abstractions must map these input and desired output values to the abstractions used in the experience abstraction process.

To summarize, a learning problem consists of an instantiation of a learning problem class, a set of experiences, a learning system with parameters suitable for the problem at hand, and a specification of the input and output abstractions, which represents a connection between the learning problem class and the experience abstraction. The learning problem definition for the example presented in this sections is shown in Listing 6.3 on page 109.

5.4 Summary

When programming with RoLL, all parts of a learning process are specified declaratively. At some point, the learning process then has to take place. Figure 5.3 on page 66 il-

lustrates the cooperation of learning problem specific and learning problem independent components defined in RoLL and how the learning process takes place.

Before anything can be learned, experiences must have been acquired by running the plan `acquire-experiences` during program execution and thereby observing the necessary experiences. The experiences are defined by an experience automaton, which is anchored to the control program execution. The robot can either be controlled by the normal program or by one using the problem generator for parameterizing the program. After one episode has been identified, the experience is converted to more abstract ones.

After that, the learning process is performed by calling the function `learn`, which executes the following steps:

1. convert experiences to learning data of the chosen learning system,
2. invoke the learning algorithm with the data and the chosen parameters,
3. integrate the learning result into the RoLL program.

After the learning process has taken place, the learned function is loaded each time the LISP environment is loaded, provided that the generated function has been written to a directory where it is loaded automatically.

5.5 Related Work on Programming Data Acquisition and Learning Capabilities

Two fundamental contributions of RoLL are that learning problems are executable and its sophisticated mechanism for experience acquisition. In the following we first present related work on experience and data acquisition in different contexts and then give a survey of languages that incorporate some form of learning.

5.5.1 Experience Acquisition

Learning experiences draw their data from two sources: from observations of the outside world and from inside the control program. The former is also practiced in observation and state estimation tasks, the latter is often necessary for debugging purposes and system evaluation.

Let's first have a look at existing systems that observe the environment. When complex processes are to be monitored, often a variety of sensors is used, which are interconnected in sensor networks. Gehrke and Madden (2004) and Madden et al. (2005) work on declarative languages that allow a specification of which data is needed and in which time intervals it is required. Because the sensors are often small and fragile and the bandwidth for communication is restricted, the focus of this research is on generating efficient plans for answering SQL-like queries about the environment observed with the sensors.

In contrast to our experience acquisition, the sensor networks don't have to take care of the episodes when the data is to be acquired. When the query is asked, the network tries to answer it as soon as possible or does so in certain intervals. But there is no need to recognize situations that are particularly interesting to observe.

The recognition of situations was examined by Dousson, Gaborit, and Ghallab (1993) and Dousson (1994). Their objective is the surveillance of real-world processes such as failure situations in airplanes. They introduce the concept of *chronicles*, which are temporal constraints on the world state. An interesting observation is that a common temporal logic representation of chronicles can be used to define situations and to represent plans (Ghallab 1998). This concept is implemented in a system called IxTeT (Laborie and Ghallab 1995). The temporal logic works on discrete events rather than a data stream. Situations are described by event patterns and temporal constraints on them. This allows to define very specific situations declaratively.

The acquisition of data from inside the program is in some way simpler, because the data is unambiguously there and no misinterpretation of sensor values is possible. On the other hand, a program is usually not written for observation. The processes and local variables are used, but not displayed. Therefore in practice there are two possibilities to collect the data from within the program: (1) to protocol continuously every variable that might be of interest for learning during the program execution or (2) to add special code to the control program specifying which data is to be recorded at which times.

The first approach is put into practice in the XAVIER project (O'Sullivan, Haigh, and Armstrong 1997) of Carnegie Mellon University. Here all the data is recorded at runtime and replayed for analysis later. However, the execution context of the program is lost when the data is replayed. This allows only an outside view of the robot by way of the control commands, but it cannot explain why the robot came to the decision. Besides, for longer runs of the robot and many variables to be observed, the amount of data gets enormous. To filter the data needed for learning is then a problem of its own.

The Common Lisp Instrumentation Package (CLIP) (Anderson et al. 1994) is a LISP package facilitating the second approach, i.e. add extra code for every kind of experience. Its goal is to provide a standardized framework for data collection, where the functionality of the program is clearly separated from the data collection. This separation, however, only goes as far as the data collection code is clearly identifiable, whereas it is still inside the actual program code. CLIP is not addressed especially for learning data, but for any kind of experiments like debugging or giving user feedback. Although the data is specified declaratively and the acquisition code is easy to add and delete from the code, it still resides inside the program and has to be added at the appropriate places in the code. When using this approach, we would have to decide a priori which experiences are needed for learning. When a new learning problem is added, additional data acquisition code must be included at the appropriate places.

5.5.2 Learning Capabilities in Programming Languages

There are only few projects where the issue of combining programming and learning is addressed. Thrun (2000) has proposed a language CES offering the possibility to leave "gaps" in the code that can be closed by learned functions. Besides the learning capabilities, CES supports reasoning with probabilistic values and the gradient descent learning algorithm implemented in CES computes probabilistic values. The main motivation for CES was to allow a compact implementation of robot control programs instead of explicit learning support. Therefore, CES only uses a gradient descent algorithm and doesn't offer explicit possibilities to integrate other learning algorithms. Besides, the training examples have to be provided by the programmer, experience acquisition is not supported on the language level (Thrun 1998b).

Andre and Russell (2001) and Andre (2003) propose a language with the same idea as CES of leaving some part of the program open to be replaced by learning. In this case reinforcement learning is used to fill in the choices that are not yet specified by programming. Since this work only considers reinforcement learning as the only learning technique, the issues of experience acquisition and program operation get straightforward: The agent executes the program, when it encounters a choice that has to be learned it selects one option according to the current rewards assigned to actions and the exploration/exploitation strategy, watches the reward to be gained by this choice and adapts its reward function on actions. Although programmable reinforcement learning agents are a powerful approach to integrate reinforcement learning into the normal control flow, it cannot be generalized to other learning techniques.

The language IBAL proposed by Pfeffer (2001) is motivated by representing the agents belief in terms of probabilistic models. Bayesian parameter estimation and reinforcement learning are offered as an operator in such a program. Markov Decision Processes (MDPs) are defined explicitly and declaratively and they can be solved by updating the reward after every run similar to the approach by Andre. The focus of IBAL is not on learning in general, but on programming with probabilistic models. The learning is merely an additional operation and only supports a certain class of learning algorithms.

DTGolog (Boutilier et al. 2000) is a decision-theoretic extension of Golog. Like in IBAL MDPs are specified explicitly and the solution of them is left to the program. The space of policies can be restricted by programming, so DTGolog supports a very close interaction between programming and learning. Boutilier et al. also emphasize that the best results can be obtained by a smooth interaction of programming and learning compared to learning or programming alone.

Chapter 6

Evaluation

We have proposed a programming language supporting the integration of learning into control programs for autonomous robots.

In this chapter we will make clear that combining learning and programming in general and the concepts of RoLL in particular are an important contribution to successful robot learning. In Chapter 1 we made some assumptions about how learning problems for autonomous robots should be designed. We state that robots should learn simple control decisions instead of huge, monolithic problems. They should further observe experience during their normal activity and use them economically. Besides, we are convinced that the best performance can be gained by combining learning and programming.

To this end, we designed the language RoLL, which provides declarative specifications for all parts of learning. It offers modular extensions to make it applicable for arbitrary learning algorithms and problems. The constructs that initiate the experience acquisition and the learning process allow a flexible interaction of programming and learning. Moreover, the experience acquisition methods of RoLL can be used for other applications than learning, for example the evaluation of plan performance.

The main focus of our evaluation lies on presenting a comprehensive example of how to specify and execute a learning problem in RoLL. Apart from that, we present some empirical results of learning problems we have solved with RoLL. In these problems, we don't want to present the quality of the learning result, because it depends largely on the choice of the experience abstraction and learning bias and doesn't mirror the merits of using RoLL. Rather, we demonstrate some advantages of embedding learning capabilities into a control language instead of performing the learning by hand.

Because of its sophisticated experience acquisition concept, RoLL is also useful for other applications than learning. We show some examples in the field of transformational planning, including cases where experience specifications are generated automatically.

As robot learning is an active field of research, we attach great importance to the extensibility of RoLL. We point out how new findings in the respective fields can be used

in RoLL and how it can contribute to explore new methods in robot learning.

Before starting the evaluation by presenting code and empirical results, we summarize the points we want to emphasize in this chapter.

6.1 Evaluation Criteria

Integrating learning into robot control languages provides several advantages that are obvious from a software engineering point of view. First of all, when a learning process is described in terms of a programming language, it can be executed automatically. Next to the comfort gained by this, the whole process can be repeated at a later time or be used on different robotic systems. For example, learning prediction models of navigation routines is important for many kinds of robots. The control signals for the robot's velocity are similar for most robots, so that the learning problem implemented for one specific robot can easily be carried over to another one.

Furthermore, when learning is included in the language, the interaction between programming and learning can be achieved in a very natural way. Functions that are best be learned, because they rely on environmental conditions, can be combined with programmed heuristics, which are often hard to incorporate into the learning process directly.

If learning is executable and specified explicitly, even more ambitious goals can be achieved in the long run. One problem in learning is that the result depends strongly on the parameterization of the learning algorithm. This choice could be performed by meta-learning algorithms (Abraham and Nath 2000; Abraham 2003; Younger, Hochreiter, and Conwell 2001) or heuristics. The same is true for preparing the experiences with good abstractions (Bonnlander 1996; Herrera et al. 2006). Besides, the execution of the learning problems can be scheduled automatically, for example considering the idle times of the robot, so that problems with higher priority are learned first.

RoLL is distinguished by a declarative, compact syntax. In the next section we show examples demonstrating this point. Although we cannot present statistically meaningful tests, we can claim that RoLL is easy to understand and use. Several students have used it successfully, in some cases without any instruction except some examples.

Moreover, RoLL separates the code for learning, especially that for acquiring experiences, from the normal program code. The learning specifications don't affect the rest of the program, so that learning problems can be added and removed without changing anything else.

Finally, RoLL is very flexible. The learnable program parts, experience storage types and learning systems can be added in a modular way. This means that RoLL doesn't depend on specific external programs or system settings and can be adapted to new developments in robot learning.

Although an empirical evaluation of these benefits is hardly possibly, we demonstrate

some practical applications of RoLL in Section 6.3 and thereby justify the assumptions we made about robot learning in Chapter 1. In particular, we stated that robots should learn simple problems rather than complicated, monolithic ones, that the experience should be observed during normal execution, and that the combination of programming and learning yields better results than learning or programming alone.

We show how a set of raw experiences can be used for several learning problems or for different solutions of the same learning problem, thereby demonstrating how experiences are used economically. Also, RoLL allows the comparison of different learning algorithms and parameterizations, which facilitates their manual adjustment and enables the use of meta-learning methods. We further show that experience observed during normal activity is more useful than artificially acquired one. In addition, we provide an example where learning and programming are combined. The presented examples should give a flavor of how important learning is for intelligent autonomous robots to be adapted to their environment, although this point has already been emphasized by others (Thrun et al. 2006; Riedmiller and Gabel 2007; Petkos, Toussaint, and Vijayakumar 2006).

The best way to show the benefits of a programming language like RoLL are programs written in the language. To appreciate the modularity and simplicity of the declarations, we present a complete example of a learning problem specification and how the learning is integrated into the program.

6.2 Comprehensive Example

The following example of a complete learning problem including experience specifications assumes that the RoLL core language has been supplied with the learning problem class, the experience class for storing experiences in a database, and a learning system for regression tree learning. The definitions of these parts are explained in Appendix D.

The problem to be learned is a time prediction model for a navigation routine, which is called `b21-go2pose-pid-player`. It takes the robot from its current position to a goal position, which is specified by 2D coordinates and an orientation. In our robot implementation there is a state variable `timestep`, which gives the time as an integer based on LISP's internal real time and a scaling factor according to the speed-up of the simulation.[1]

We present the learning process according to the steps in Figure 2.2 on page 19.

6.2.1 Raw Experience Acquisition

For accessing the time needed for a navigation task, the navigation routine must be observed to acquire a raw experience like the one defined in Listing 6.1. The experience

[1] This means that if the simulation is sped up by a factor n, the internal time is also accelerated by n, thereby ensuring that the program reacts identically for different simulator settings.

automaton is anchored to the control program by defining the activity of the routine `b21-go2pose-pid-player` as an episode (line 4). For describing the task at hand, the start and goal positions must be known (lines 6–8). In addition, the time stamps of the starting and stopping time points are recorded (lines 5, 9). The goal position is stored in the routine description, which is bound in the lexical scope of the RPL task corresponding to the navigation routine (line 8).

Once an experience has been recorded, the specification tells RoLL to check whether the observed process has been aborted or has failed. In this case, the experience is not used for learning (lines 10,11).

The experience can be identified by its name `navigation-time-exp`. All experiences are stored in a global hash table and can be retrieved with the function `getgv` (get global variable). For observing instances of the specified experience, the RoLL command `acquire-experiences` is called with the desired experience object.

The definition of raw experiences is very explicit and declarative. The anchoring to the program is established clearly and also the data associated with the episode is well-structured and understandable. The simplicity of the specification enabled us even to generate experience definitions automatically in the context of transformational planning in order to parallelize the execution of multiple plans (Bachmann 2007). In spite of the simplicity, complex, hierarchical experiences can be specified just as easily as the example in Listing 6.1. The most complex automaton we used included 11 sub-automata and recorded over 60 variables (see Section 6.4).

For learning a model of the navigation routine, both passive and active experience acquisition are conceivable. Here we demonstrate the difference in terms of definition. In Section 6.3.1 we show an empirical comparison between the two strategies.

Listing 6.1 Raw experience definition for navigation time model.

```
1  (roll:define-raw-experience navigation-time-exp
2   :specification
3    (:anonymous-automaton
4     :rpl (:routine-execution 'b21-go2pose-pid-player)
5     :begin ((timestep [getgv 'statevar 'time-step])
6             (start-pose [getgv 'statevar 'b21-pose])
7             (goal-pose
8              (pose (goal (:internal-value "ROUTINE" :this :exact)))))
9     :end ((timestep [getgv 'statevar 'time-step])))
10  :experience-handling (((or (:event :abort) (:event :fail))
11                         :discard)))
```

Passive Experience Acquisition

Passive experience acquisition means that the robot's actions are not tailored to the learning problem. Rather, the robot performs standard activities. In this case, we use a plan for setting the table. The two processes — observation of the experience and the plan for table setting — are executed in parallel.

```
(pursue
 (roll:acquire-experiences (getgv :experience 'navigation-time-exp))
 (execute (make-instance 'set-the-table-controller)))
```

The `acquire-experiences` command can be used anywhere in the program. However, one motivation for using RoLL is not to modify the program for learning. Therefore, it is best to run the observation process in parallel to the top-level control program. It identifies the interesting parts of the execution automatically.

Active Experience Acquisition

Setting the table involves few navigation commands. Therefore the overall state space needed for a full modeling of the navigation routine can be observed only partially. A different approach is to generate the navigation tasks to be performed with the means provided by RoLL.

```
1  (pursue
2   (roll:acquire-experiences (getgv :experience 'navigation-time-exp))
3   (roll:with-problem-parameters
4    (:parameters ((x-goal :random :min 0.4m :max 3.62m :samples 50)
5                  (y-goal :random :min 0.4m :max 3.72m)
6                  (az-goal :random :max 359.9deg))
7     :relation (:indep x-goal y-goal az-goal))
8    (achieve (make-instance 'b21-at-pose
9              :pose (make-instance '2d-pose
10                    :x x-goal
11                    :y y-goal
12                    :az az-goal)))))
```

The control program navigates to different positions in the kitchen (lines 8–12). The parameters defining the goal positions (`x-goal`, `y-goal`, `az-goal`) are generated randomly (lines 4–6).

The problem generator of RoLL is only one way to implement active experience acquisition. Any function returning the necessary parameters — possibly taking into account existing experiences and the robot's overall reward function — can be used for this purpose. For active experience acquisition as it is done in most current systems, the RoLL problem generator is a simple, yet powerful tool for defining and using different parameterizations for control programs.

6.2.2 Experience Abstraction

The position values in the raw experience are absolute values. We make the simplifying assumptions that objects in the kitchen don't affect the navigation time and that the robot has means to decide if a location is accessible. With these premises, learning with absolute positions is not advisable, because navigation tasks that are only shifted or rotated on the 2D plane are treated as different cases. Therefore, we use the abstraction depicted in Figure 5.16 on page 97, which uses the distance of the two points and the angles relative to the connecting line between the two points.

In the definition in Listing 6.2, we use the construct `with-binding` for calculating intermediate values (lines 4–8). Its syntax and functionality correspond to that of the LISP `let*`. The binding of intermediate values is not necessary, but makes the definition better readable. The auxiliary values include the start and goal point from the raw experience, the duration of the navigation task, and the angle of the connecting line of the two points. Based on these values, the distance of the two points and the normalized orientations are calculated (lines 10–12).

When an instance of the raw experience is recorded, it must be processed further immediately. In contrast, we write the abstract experiences to a database, which must be specified in the experience (lines 14–17). The access to the database is performed by the experience class and is hidden from the programmer of the abstract experience. Likewise, the data is retrieved automatically.

Because the operations for the conversion are not restricted in any way, the abstrac-

Listing 6.2 Abstract experience and conversion specification for navigation time model.

```
1  (roll:define-abstract-experience navigation-time-abstract-exp
2   :parent-experience navigation-time-exp
3   :specification
4    (roll:with-binding ((p1 (:var start-pose :begin))
5                        (p2 (:var goal-pose :begin))
6                        (timediff (- (:var timestep :end)
7                                     (:var timestep :begin)))
8                        (offset-angle (angle-towards-point p1 p2)))
9     (:anonymous-automaton
10     :begin ((dist (euclid-distance p1 p2))
11             (start-phi (ut:difference-radian [az p1] offset-angle))
12             (end-phi (ut:difference-radian [az p2] offset-angle)))
13     :end ((navigation-time timediff))))
14  :experience-class roll:database-experience
15  :experience-class-initargs
16   (:database (make-instance 'roll:mysql-database
17               :host "..." :user "..." :user-pw "..." :name "...")))
```

tion is very flexible and general. As the language construct for experience abstraction is defined analogous to the raw experience specification, it organizes the data in a natural way while enabling complex structures for abstract experiences. The modular storing mechanism, which in particular offers the storage of experiences in a database, enables experiences to be understood more thoroughly and to be used not only for learning, but also for program debugging and evaluation.

6.2.3 Learning Problem Definition

Finally, we define the learning problem as shown in Listing 6.3. The function to be learned is the time model of the routine `b21-go2pose-pid-player` (line 2). As an experience we use the same abstraction as the one presented in the last section (lines 5–10). But the experience type of the former was a database experience, which doesn't comply with the format required by the WEKA learning system. Therefore, the experience used for learning is defined to be of class `weka-experience` (line 11). The WEKA experience type requires the specification of the WEKA types for the input and output variables (lines 12–16).

As a learning system we use the WEKA M5' algorithm, which supports model and regression tree learning. Beside some path specifications (lines 18–21), we adjust the algorithm to learn a regression tree instead of a model tree (line 22).

The input to a routine model is always the routine object addressed by the variable `routine`. For calling the learned function, the same abstractions as for the experiences must be performed. In the learning problem specification we inform RoLL that this abstraction can be used for calling the learned function, but in the raw experience definition the variable `goal-pose` is not obtained from the local variable of a certain process, but should be set to the goal pose as contained in the routine variable given to the model (lines 23–25). The result value of the regression tree is exactly what the learned function should provide, so no conversion is necessary (line 26).

Like the experience specifications, the structure of the learning problem is very explicit. The aspects influencing the learning result (experiences and learning system) are separated clearly from the ones specifying the program part to be learned (function, input-conversion, output-conversion). Therefore, the parts controlling the success of the learning process can be modified and adapted by meta-learning algorithms (for the learning system) or by automatic feature selection (for the learning experience).

To initiate the learning process, the learning problem must be executed by calling the function `learn`:

```
(roll:learn (getgv :learning-problem 'b21-go2pose-time-model))
```

After the learning has been completed, the learned model is loaded at once and then every time with the rest of the robot program and used for example as a time-out criterion

for the execution of the navigation routine.

The call of the `learn` command can be added at arbitrary places of the code. This enables the learning of multiple, interacting problems by specifying an order in which the problems are to be learned. Besides, the robot's primary activity can be considered. For example, idle times can be used for performing the learning tasks. When idle times are identified automatically, the learning can be added to the program by plan transformations (for a similar transformation problem see page 123).

6.3 Empirical Results

In the following we present some empirical results from learning problems we have solved with RoLL. The examples demonstrate some advantages of automatic learning instead of manual implementation or learning huge monolithic tasks and show that our approach of learning small problems with passive experience acquisition on autonomous robots is a

Listing 6.3 Learning problem definition for navigation time model.

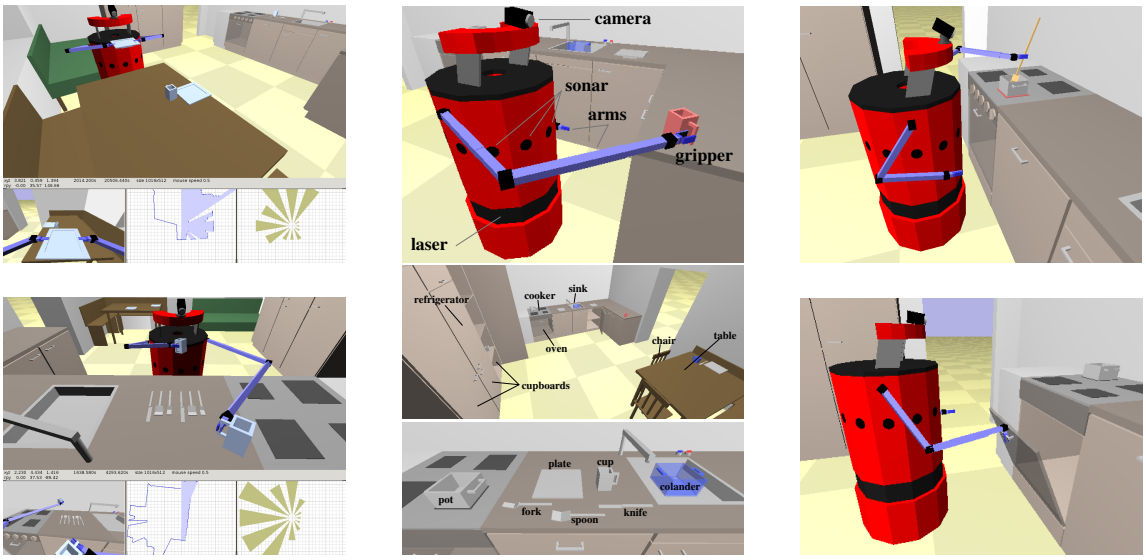
```
(roll:define-learning-problem
 :function (:model b21-go2pose :time)
 :use-experience
  (:parent-experience navigation-time-abstract-exp
   :specification
     (:automaton
      :begin ((dist (:var dist :begin))
              (start-phi (:var start-phi :begin))
              (end-phi (:var end-phi :begin)))
      :end ((navigation-time (:var navigation-time :end))))
   :experience-class roll::weka-experience
   :experience-class-initargs
    (:attribute-types '((dist numeric)
                        (start-phi numeric)
                        (end-phi numeric)
                        (navigation-time numeric))))
 :learning-system (roll:weka-m5prime
                   :root-dir (append *root-dir* '("learned" "src"))
                   :data-dir (append (pathname-directory
                                      (user-homedir-pathname))
                                     '("weka"))
                   :build-regression-tree T)
 :input-conversion (:generate
                    (:in-experience navigation-time-exp
                     :set-var goal-pose :to (pose (goal routine))))
 :output-conversion (navigation-time))
```

promising one.

For our experiments we used a simulated household and robot based on the Gazebo simulator[2] (Beetz et al. 2007). 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 expensive and hard to maintain. The simulation is much cheaper, better available and more flexible concerning different robot hardware and different environments. 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 is necessary, in which we are not particularly interested. Therefore we added automatic doors to the kitchen that can be remote-controlled by the robot. This is an assumption that could very well be fulfilled in a real kitchen, but the simulation could be implemented with much less effort.

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

Figure 6.1 Realistic simulation of a household robot working in a kitchen using the Gazebo simulator.



[2] http://playerstage.sourceforge.net/gazebo/gazebo.html

robot. This is possible with Player (Gerkey, Vaughan, and Howard 2003), which makes available a device-layer providing a network interface to the hardware (or simulated hardware) underneath. This enables the use of 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 (2D Cartesian coordinates and orientation) are given as percepts (which is quite realistic for a laser-equipped robot in a known environment) and the position of all objects in the robot's field of view can be determined accurately as 3D Cartesian coordinates and 3D orientation. The simulation is 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 makes the simulation very convincing.

The following experiments were performed on the simulated kitchen robot shown in Figure 6.1. Its most sophisticated plans comprise setting the table and boiling pasta. The learned functions were fully integrated into this control program and are used in our current robot program.

6.3.1 Time Model of the Navigation Routine

We have already shown the code for defining the learning problem for a time prediction model of a navigation routine. Here we present two experiments for this problem. The first one compares different learning algorithms for the problem, the second shows that passive experience acquisition can be more appropriate than active acquisition for robots in real-world environments.

Comparison of Learning Systems

As a reference for the quality of the learning result we use a programmed function. We have applied this programmed function for some time for determining the time-out criterion when the routine is called. For this application its accuracy is absolutely sufficient.

For our experiment, we generated a set of 20 test navigation problems (randomly with RoLL's problem generation facilities) and chose three learning systems (neural network, model tree, and regression tree). Then we started with an empty set of experiences and in each cycle added 40 actively acquired, randomly generated experiences to the set of training experiences. After each iteration, the function was learned with each learning system and the learned functions predicted the time needed for the test navigation problems. Then the average of the relative error of the prediction was calculated for each learning system.

Figure 6.2 depicts the result of this experiment. The first interesting observation is the prediction error of the programmed function. As the same 20 test problems are used for

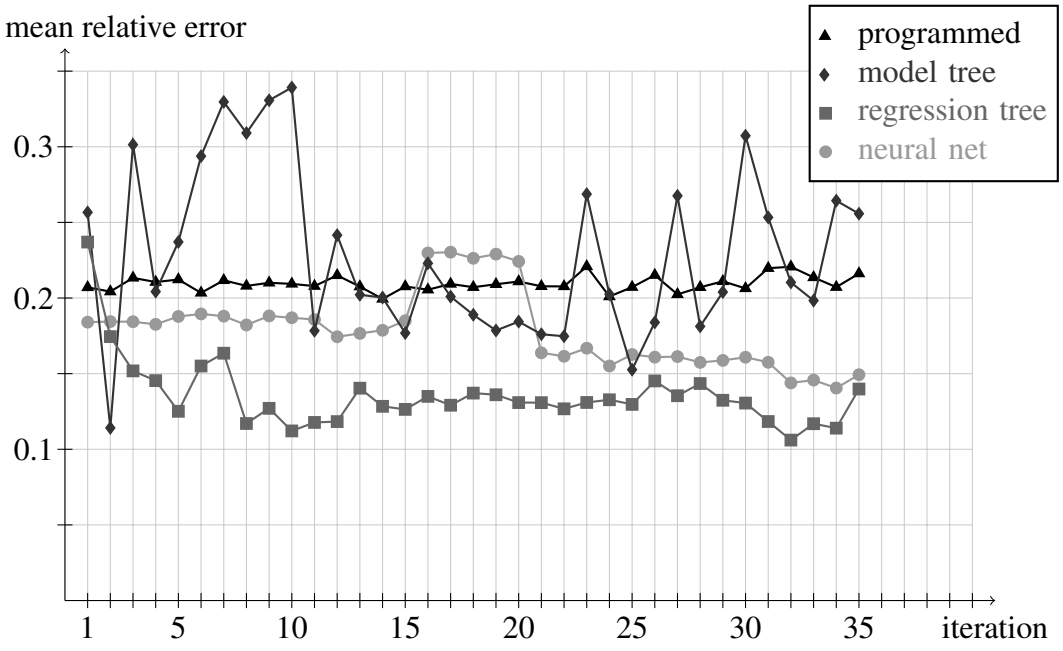
all evaluations, this line should be straight. However, we had the robot perform the test problems for each evaluation. Since our simulation is non-deterministic the navigation times differ, but the predictions are the same.

The first observation is that all learning systems produce results in about the same range of accuracy as the programmed function. The learning result of the model tree varies considerably in the number of experiences provided. The results of the regression tree is the less surprising. With a small number of examples, the error is high and gets small with more experience. Training with more than 400 experiences doesn't improve the result further. The development of the results of the neural net is very unexpected, because with only 40 experiences it is as good as with 400. One explanation is that the figure only shows an average value. In figure 6.3 the development for only one test navigation task is shown. The results are much more what one would expect.

This experiment is not meant to make any general statement about the different learning algorithms. It is only intended to show that the learning result depends on the used algorithm and that the systems and their parameters can easily be reconfigured with RoLL.

For testing the different learning systems, only the learning problem had to be defined several times. Apart from comparing learning systems, different parameterizations of one learning system (e.g. the structure and learning functions of a neural network) can

Figure 6.2 Prediction error relative to needed time for navigation routine comparing different learning systems.



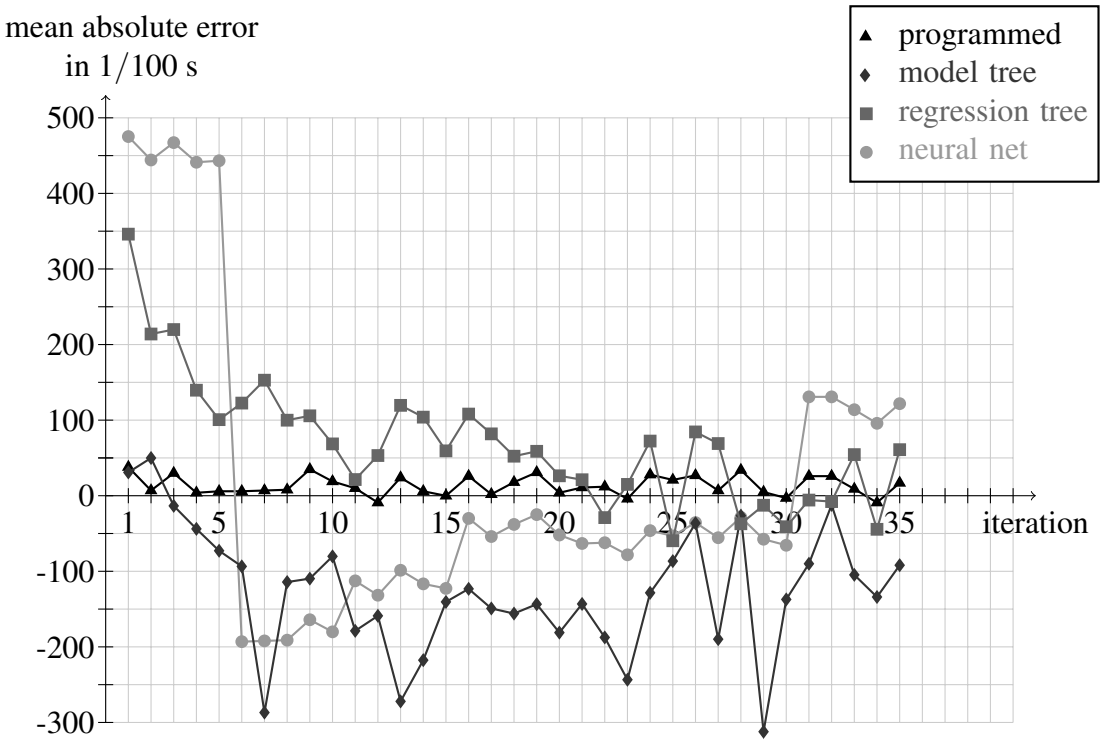
also be contrasted, which works exactly the same way. Besides, the RoLL constructs for learning can be used for evaluation, too. For generating the test problems, we could resort to the RoLL problem generation and observe the time required for this navigation tasks with the normal experience acquisition techniques. In all, these is the functionality needed for meta-learning: learning the same problem with different learning systems or parameterizations and comparing them. Because of the explicit structure of the learning problem definition, the parameters for the learning system can be adapted automatically.

Getting examples from the real-world distribution

Currently, our household robot can only perform a few high-level household activities such as preparing pasta or setting the table. If these high-level plans only contain a small subset of all possible navigation tasks, there is no need to learn a model for the whole state space. Therefore, we compared the learning performance using an artificially (actively) acquired set of experiences and one that was observed while the robot was performing one of its high-level tasks.

As high-level plans we used setting the table, which involves seven navigation tasks,

Figure 6.3 Absolute prediction error for one test sample.



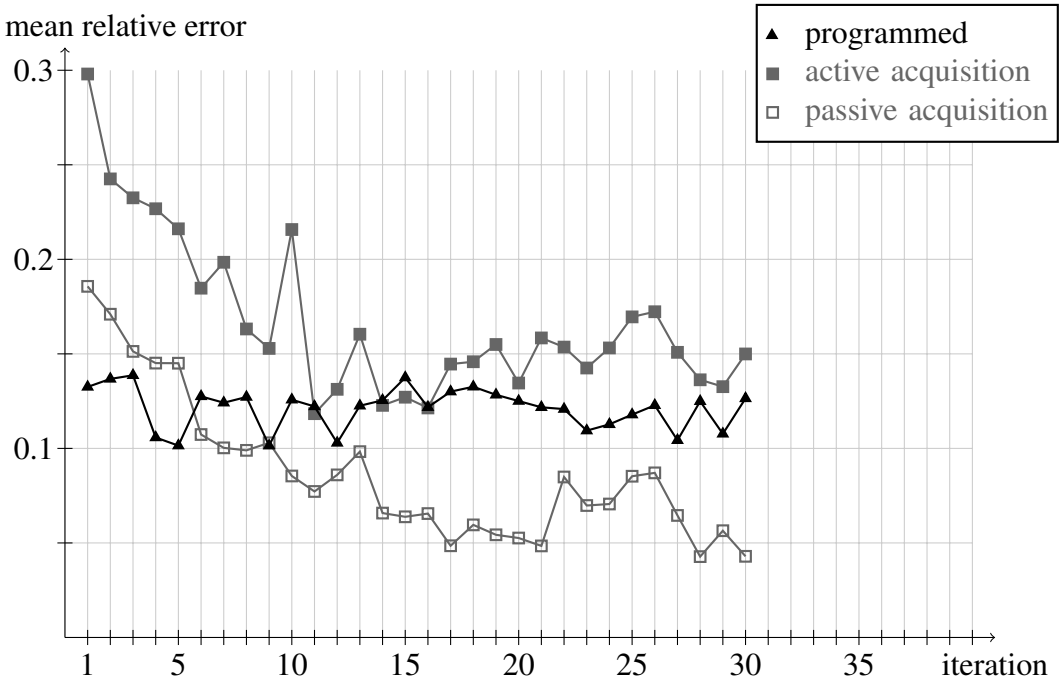
and a plan for boiling water, which is like cooking pasta, only without pasta (of course, this is not really useful, but suffices for our purposes), involving eleven navigation tasks. As a learning system we chose a regression tree, because it performed well in the last test.

The experiment was similar to the one for comparing learning systems. In each iteration, the set of experiences was extended by one[3] observation of the navigation tasks in the table setting plan and 30 actively acquired experiences respectively. Then the table setting plan was executed once and the time predictions of the two learned and the programmed function were compared with the observed navigation times.

Figure 6.4 shows the results. It is obvious that the function trained with experience observed during plan execution performs much better than the one with the general experiences, although it uses less learning data in each iteration.

Interestingly, the regression tree trained with artificial experiences seems to be worse than the one from the previous experiment (c.f. Figure 6.2). This is probably due to the fact that the navigation tasks in setting the table are very similar, so that the overall error mirrors the error on this class of navigation tasks and is therefore more comparable to the

Figure 6.4 Prediction error relative to needed time for navigation routine comparing active and passive experience acquisition.



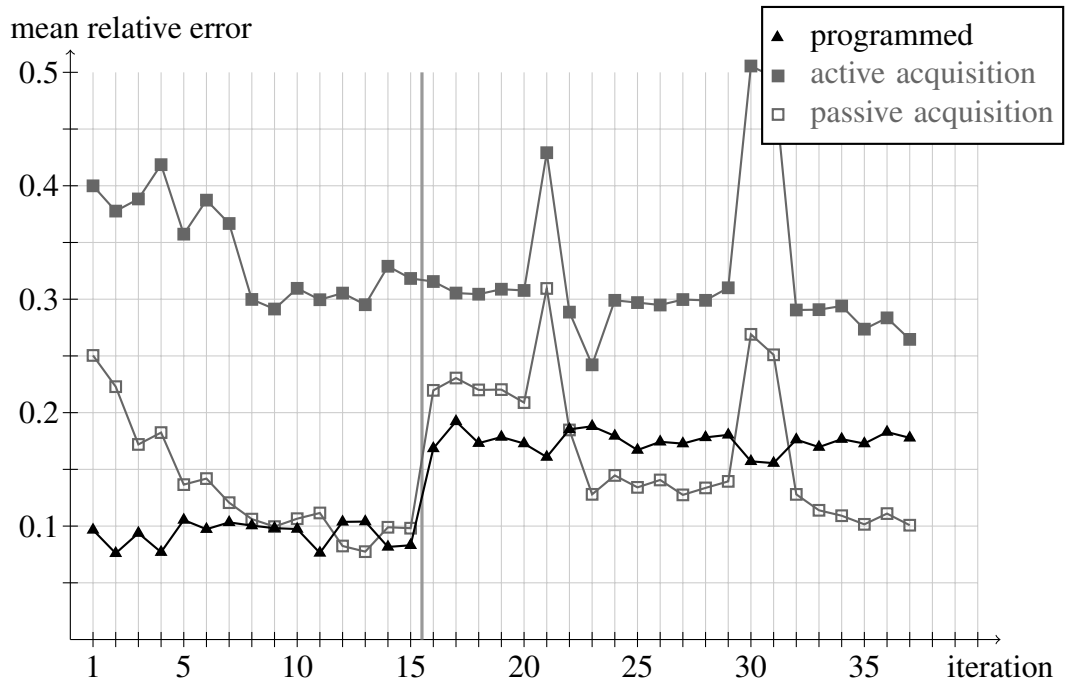
[3] In the first iteration the plan must be performed twice, because the regression tree algorithm needs more than 10 experiences to work.

single case of Figure 6.3. Besides, in the previous experiment, in each iteration 40 new experiences were added, while this time we only added 30.

What we did when learning with experience drawn from the table setting plan is pure overfitting of the function. We cannot expect it to work properly for other plans in the household. But we wanted to know how much work is needed to adapt the function to another plan. Therefore we first repeated the experiment shown in Figure 6.4 for 15 iterations, then we changed the plan from table setting to boiling water and iteratively acquired experiences and tested the functions on the boiling water plan. Figure 6.5 illustrates the results.

As expected, the prediction error of the specialized function is higher for the water boiling plan than for the table setting plan and decreases with more experiences from the water boiling plan. Also, the performance of the the function trained with actively acquired experiences isn't affected by the change in the test set. Surprisingly, even without any experience from the water boiling plan, the function learned with "natural" experiences is still more accurate than the one covering the whole state space. An explanation could be that the navigation tasks in the table setting and water boiling tasks are similar and probably are for most navigation tasks in a kitchen, which is an indication that passive

Figure 6.5 Prediction error relative to needed time for navigation routine comparing active and passive experience acquisition with different plans.



experience acquisition can outperform active acquisition, because large parts of the state space aren't needed for good predictions.

This problem demonstrates that learning on the job is often more convenient for robot problems than generating artificial problems to gather experience from. There is no reason for a robot to learn about situations that it never encounters. Rather, it should concentrate on the problems it is faced with every day. This makes learning problems more manageable and successful.

6.3.2 Optimized Gripping Procedure

The second experiment is designed to demonstrate how RoLL provides deeper understandings of learning problems and how learning problem definitions can be adapted and optimized easily, a requirement for enabling meta-learning. The task was to learn which hand the robot should use when gripping a cup. The raw experience definition for this experiment is depicted in Figure 5.12 on page 88 and includes the original cup and robot locations, the goal position of the cup (when acquiring experience the cups were gripped and put down at another position), the position where the robot placed the cup and the times when the gripping and put-down routines started and stopped.

For deciding which hand to use for gripping, one might only consider the situation before the gripping takes place (which is what we do in the programmed function used as a reference for comparison). In fact, we first learned the function only considering the initial gripping position and the time it would take to grip the cup. But we suspected that when the cup is put down later, the hand influences the performance, too. This is why we gathered more data in the raw experience than we needed for the first learning trial.

Choosing a Hand for Gripping

Instead of learning a function deciding on the hand to use, we learned time prediction models for gripping the object with each hand. When a hand is to be chosen, the predicted gripping times for both hands are compared and the faster side is chosen. This approach is more flexible than learning the decision function directly. For example, there might be more information in specific situations, e.g. that one of the hands can grip a certain kind of object more reliably than the other. The choice of the hand can then be extended from pure time considerations to other aspects.

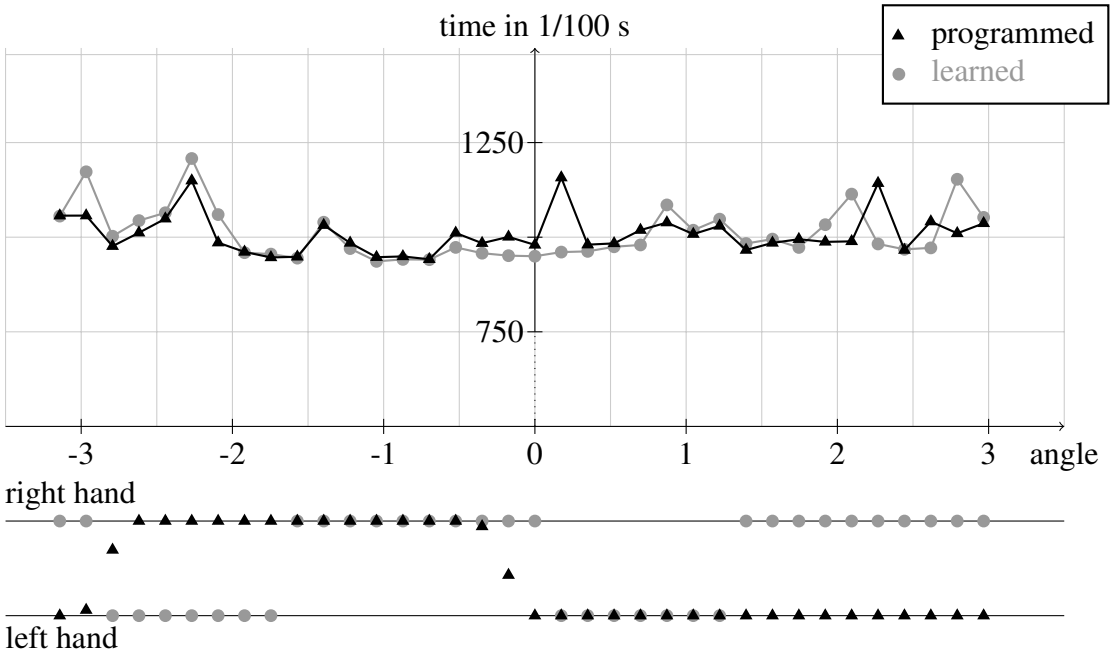
The abstracted experience only includes the distance between the robot after it has positioned itself in front of the cup and the relative angle of the cup with respect to the robot. As learning system we used neural networks.

The resulting decision function was evaluated against a programmed one, which we have been using successfully for some time. We put the cup and the robot at fixed positions and varied the position of the cup handle relative to the robot (i.e. we varied the angle

of the cup handle)[4]. All gripping tasks were executed 10 times for avoiding accidental outliers. Figure 6.6 shows the results of this experiment. The graph at the bottom indicates which hand was chosen by each routine for the given tasks[5]. In many cases, the two functions don't agree in their choice. The graph above shows the time needed to grip the object with the chosen hand. Here we see that the task always takes about 10 seconds, no matter which hand is used. So there is hardly any difference in performance between the two functions.

This evaluation shows that the hand used for gripping a cup affects the performance only minimally. Therefore, we learned new models that consider not only the gripping task, but also that of putting down the object. To do this, we only had to define a new abstraction of the raw experience, which already included the information we needed, and to define new learning problems for the time models of each hand.

Figure 6.6 Comparison of functions for deciding which hand to use considering only the pick-up task.



[4]We also varied the relative positions of the robot and the cup, but these results are hard to be displayed graphically. The results were comparable to those shown in Figure 6.6

[5]Our programmed function considers more parameters than the turning angle of the cup shown in the graphic, so that for some angles it chooses different hands in different runs. The graph shows the "average" hand chosen on a scale between 0 and 1.

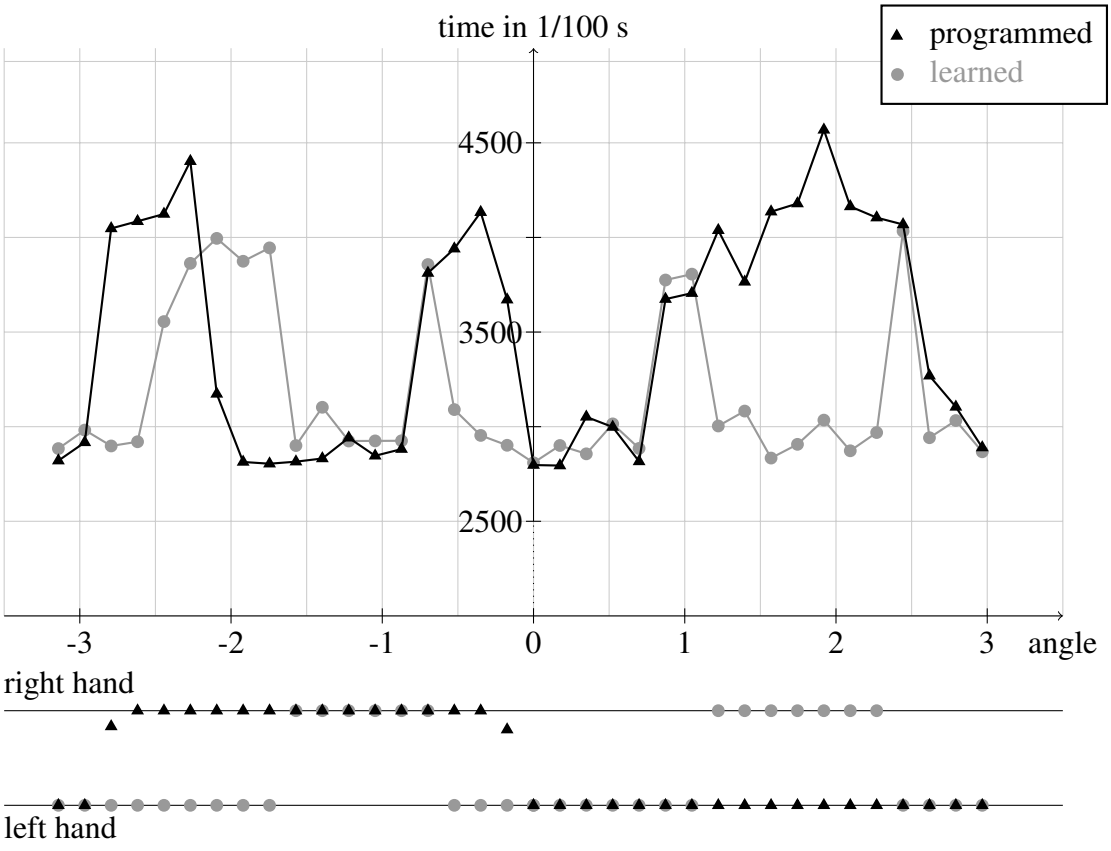
Considering the Next Action

In the second attempt we used the information of gripping and putting down the cup for selecting the hand. As in the first trial we trained two neural networks, one for each side, predicting the time needed to fulfill a task. The chosen abstraction was even simpler, we used only the relative orientation of the cup with respect to the robot for the gripping and the putting-down tasks. The position of the robot when it is gripping or putting down an object is selected by a function and therefore the distance to the cup is usually similar.

Again we evaluated the result against the programmed function, which is a little unfair, because the programmed function doesn't consider the putting-down process. On the other hand, it was very easy to learn the new models, whereas we would have had to do a lot of parameter tweaking for upgrading our programmed function.

The bottom graph in Figure 6.7 shows that again the two functions returned different results in many cases. But this time, the choice has a much higher impact. As the initial

Figure 6.7 Comparison of functions for deciding which hand to use considering both the pick-up and put-down tasks.



position of the cup doesn't matter much in the choice of the hand, this time we depict the different turning angles of the cup's goal positions. The times needed this time are higher than before, because we observe both the gripping and putting down processes. The time differences observed by using different hands often amount to about 10 seconds and most of the times the choice of the learned function is to be preferred.

With this second learned function, we performed another, more realistic test. We made the robot set the table for four people using only cups and recorded the time needed to complete the overall task. We defined ten initial situations, where the cups were kept at different locations. For each situation, the table had to be set several times to avoid incidental results caused by variations in the simulation. The table setting with the programmed routine needed about 4.32 minutes on average, whereas the average for the learned one was only 3.85 minutes. Table 6.1 shows all the observed times for each initial situation and the time difference.

With this learning problem of choosing the hand for gripping, we want to emphasize several advantages of RoLL. We have shown that the learned function performs at least as well, usually better, than the programmed one. When we discovered that the putting down task should be considered, the learning and experience specifications hardly had to be changed. Compared to reprogramming the manually implemented function, the effort was minimal. Besides, this problem is an example of how experiences can be used for different versions of a learning problem. This technique can be applied equally for distinct learning problems that need similar experience data. Finally, we have argued that the combination of programming and learning is more powerful than just learning alone

Table 6.1 Comparison of learned and programmed hand choosing function when setting the table with four cups. The times for the two functions are an average for all runs from the same initial situation, given in 1/100 s.

situation	programmed	learned	time difference
1	24861.3000	22509.7000	2351.6016
2	25288.3000	24568.2000	720.1016
3	28470.5000	23663.2000	4807.3010
4	26318.2500	22655.2000	3663.0508
5	28813.3333	21627.6364	7185.6953
6	24210.6000	21488.0000	2722.5996
7	23360.6667	21916.3000	1444.3672
8	27907.5000	24397.4615	3510.0370
9	25610.1538	23699.8182	1910.3359
10	24232.8000	24251.0000	-18.1992
Avg	25907.3400	23077.6516	2829.6891

(because it is inflexible) or programming alone (because it performs worse).

6.3.3 Choosing an Arm Trajectory

Next we present evidence of how difficult the handling of experiences and the choice of a set of experiences for learning can be. The modular structure of RoLL is used to identify properties of the experience set, which is necessary to formulate problems adequately.

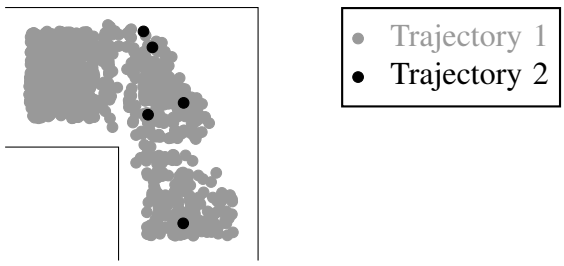
A problem in our kitchen was that gripping objects in the corner of the kitchenette often failed, because the arm got stuck when the robot tried to move its arm from its navigation pose (where the gripper is below the worktop) to a position above the worktop. To improve this, we defined two trajectories for the arm to move. The question was then, which trajectory should be used in which situations.

Because the arm trajectory depends on which arm to use and vice versa, the arm to be used depends on which trajectory would be applied to each arm, the problem was tackled in a combined approach. For observing experiences, a cup was set at a random position in the range of the corner. In each situation all four possibilities of gripping (two arms and two trajectories for each side) were performed. We recorded the time it took to perform the task or when the gripping failed, which kind of failure had occurred. For each experiment we chose the fastest combination of parameters as the target value.

With these experiences we trained a decision tree, but the result was disappointing. Figure 6.8 shows the reason. When only the trajectories are examined, without considering the arm, the simpler trajectory works in almost all cases. A decision tree ignores the very few examples for the more complicated trajectory considering them as outliers.

The question is now, why the gripping was so bad in the first place. Presumably the general function for selecting the arm doesn't work properly in the corner. The trajectory seems to be only of secondary impact. For being sure that the robot can grip objects at all positions the most stable reaction would be to add a failure recovery step that tries to use

Figure 6.8 Experiences for choosing the trajectory. The dots indicate the cup's position, the orientation is not shown, but was chosen randomly, too. The areas near the table edge were banned as possible cup positions.



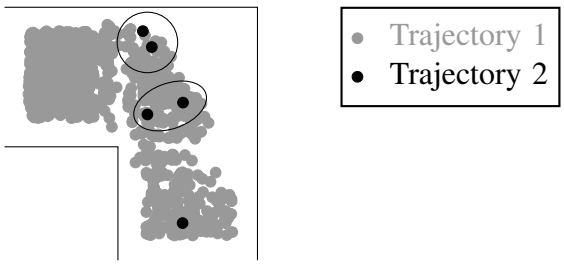
the more complicated trajectory if the simple one didn't work.

Another possibility would be to select the learning experience in a different way. Kirsch, Schweitzer, and Beetz (2005) have shown that the best set of experiences cannot be determined when considering the experiences in isolation. Rather, the whole set of experiences should be taken into consideration. In Figure 6.9 one can identify two areas where the more complicated trajectory might be more useful than in other situations. When all positions in this area can be reached with Trajectory 2 without much efficiency loss, they should better be considered as experiences for using Trajectory 2. The learning algorithm would then be able to identify the regions where the complicated trajectory is the safer choice.

As in the learning problem for selecting which hand to use, RoLL helped to find out that the problem was formulated wrongly. There are ways to improve the gripping performance in the kitchen corner, but not by choosing the arm trajectory. Not only the learning capabilities are important here, but also the insight into the experiences, which can easily be gained with RoLL.

This learning problem is also a good example for using experiences for several learning problems. When cups were placed in the far end of the corner, the robot couldn't reach them with any combination of parameters. With this information the robot can learn which positions are unreachable for it.

Figure 6.9 Candidate areas for using Trajectory 2.



6.4 Further Applications of RoLL

Although RoLL is designed primarily for performing complete learning tasks, its sophisticated experience acquisition and managing concepts allow it to be used in other contexts of robot control. Beside classical learning, robot behavior can be improved by transformational planning. The Traner project (*Tra*nsformational Plan*ner* for Everyday Activity) in our research group examines how a robot can adapt default plans to specific situations (Müller, Kirsch, and Beetz 2007). We describe several tasks in the transformational planning approach, where experiences are needed. Besides, we show how RoLL can be used

to evaluate and debug robot control programs. All these applications clearly demonstrate the flexibility and generality of the RoLL constructs.

6.4.1 Transformational Planning

In contrast to the classical planning approach, where plans are generated from abstract actions, the basic idea of transformational planning is to provide a robot with a library of default plans (Müller and Beetz 2006) for standard situations and adapt these plans to specific situations. In the approach described by Müller, Kirsch, and Beetz (2007), the robot chooses a plan from the plan library when confronted with a problem. After executing the plan, it evaluates its performance according to certain criteria like the time needed to fulfill the task. If the plan hasn't shown satisfactory results it is transformed using predefined transformation rules. The resulting plan is then tested in simulation and evaluated again. This procedure is continued until the plan performs satisfactorily.

One question in this approach is how to select promising transformation rules. One criterion is the syntactical structure of the plan. Transformation rules are defined to match only certain plan structures. Besides, the transformation rules are provided with an applicability criterion stating when the transformation is most useful. For example, when a robot performs tasks several times while executing the plan, a good transformation would be to use all the robot's resources in order to perform the task less often. One instance of this rule is when the robot carries several cups to the table while setting it, a good transformation would be to use both grippers instead of only one as in the standard plan.

So one application of experience acquisition in plan transformation is the observation of the plan execution with respect to the applicability conditions of the transformation rules. In the example, the robot would have to monitor which processes are executed using which objects. Besides, the evaluation criteria like the time needed or the number of failures that occurred during the execution must be recorded. The evaluation of the plans presented by Müller, Kirsch, and Beetz (2007) was performed using the RoLL constructs for experience acquisition and storing the data as permanent experience in a database.

An important problem in everyday environments is when to integrate auxiliary goals like opening and closing cupboard doors or cleaning up. The default behavior of a robot might be to close the cupboard every time it has taken something out. This can, however, result in very undesirable behavior. Imagine a robot that is to take two cups out of the cupboard. It opens the door, picks up one cup, closes the door, opens it again, takes out the second cup and then finally closes the cupboard again. It is extremely hard to find a general solution to this kind of problems. Plan transformations can handle these questions in a natural way by applying a transformation rule that tells the robot to close all doors at the end of the plan (which might not be the optimal solution for the whole plan, but when applied to parts of the plan can deliver impressive results). For knowing, which doors are opened, the robot must monitor its execution and remember the open doors. For this,

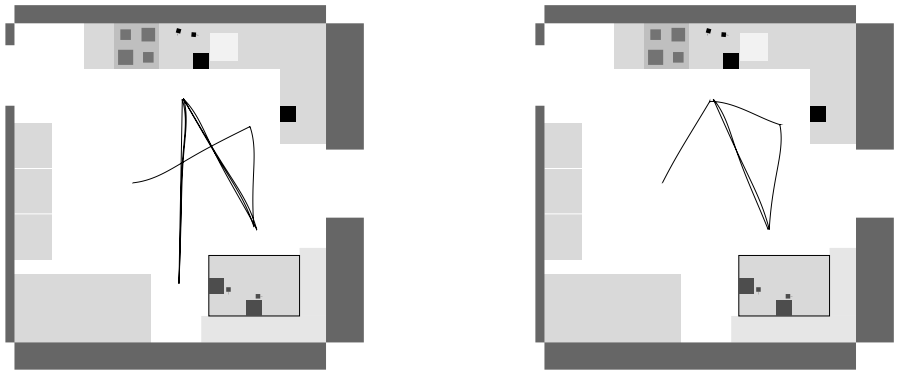
we used the experience acquisition capabilities of RoLL. The plan transformation rule automatically builds the acquisition code into the plan, the experience definition being predefined. The raw experiences are converted to an abstract experience that represents the data in a list containing the doors to be closed at the end of the plan.

Another interesting problem is how to execute a set of plans efficiently by interleaving their execution. In a master's thesis, Bachmann (2007) has investigated how idle times in one plan can be used to perform steps of other plans. As an example, the robot had to prepare pasta and set the table simultaneously. The prerequisites for doing this were (1) to have accurate models of how long plan steps will take, and (2) to have knowledge about idle times during plan execution. To make the approach general, the experience needed for these two problems was acquired with automatically generated definitions of raw experiences. First the plan was analyzed with respect to its subplans. For all subplans, raw experience definitions were generated to acquire the data for learning time prediction models. This shows that the experience definitions in RoLL are declarative enough even to be generated automatically. The models were used to transform the sequential execution of two plans into a parallel execution. A similar transformation could be defined to intertwine the learning of predefined problems with the normal execution.

6.4.2 Plan Evaluation

RoLL cannot only be used to improve programs either by learning or plan transformation, but it can help to get a better understanding of complex plans and their execution. Figure 6.10 graphically displays an episode of the robot setting the table. The picture in

Figure 6.10 Navigation trajectories of default and optimized plan. The data was recorded by the RoLL experience acquisition facilities. The original positions of the cups and plates are indicated in black, the goal positions on the table in dark gray.



(a) Default Plan. (b) Optimized Plan.

Figure 6.10(a) shows the navigation paths using a default plan from the plan library, where the plates and cups are carried to the table one by one. The improvements of transforming this plans to stack plates and carry the stack to the table, to use both arms for carrying the cups, and to navigate to a more convenient position at the table are clearly visible in the picture in Figure 6.10(b). The robot has to perform much less navigation tasks. In the same experience acquisition, the occurrence of failures and when they happened were observed.

This example is a complex instance of experience acquisition with an interesting automaton structure. The main episode is identified by the table setting task. During this episode, sub-episodes are defined by the navigation, grip and drop routines. During the navigation the robot's position is recorded continuously to provide the trajectories shown in Figure 6.10. When the gripping started, the original object positions were observed, and at the end of the dropping routine, the final positions were gathered. Besides, failures were monitored and stored with the context they occurred in.

In total, the raw experience automaton comprises three levels of hierarchy, 11 sub-automata, and protocols more than 60 variables. While the raw experience is structured along the program structure given by the routines used during the table setting task, the abstract experience is structured along abstract concepts such as failures, manipulated objects, and arm movements. In all, the abstract experience contains more than 100 variables in 7 sub-automata on two levels of hierarchy.

6.5 Extending RoLL

We have demonstrated the benefits of RoLL by providing a complete code example, presenting the results of different learning problems, and a description of applications that do not directly involve learning. It is obvious that our language offers declarative specifications of all parts of the learning problem, especially experience acquisition. All specifications have an explicit structure, so that they can be modified by automatic procedures, for example by plan transformations or meta-learning mechanisms.

The field of robot learning is an active field of research. Therefore, RoLL cannot be expected to comprise all the functionality that might be needed for robot learning in the future. All currently available mechanisms for robot learning can be used in RoLL, so that it is a self-contained system comprising all aspects of experience-based learning. However, some mechanisms haven't been fully understood, for example meta-learning, continual learning, or automatic feature detection (see Section 6.7 for an overview on the state of the art of these topics). As a consequence, RoLL provides the flexibility to be extended when new findings enable more sophisticated robot learning and can serve as a tool for exploring new methods.

Apart from the explicit extensions offered by RoLL (learning system, experience class,

and learning problem class), RoLL includes all the functionality of LISP. This means that any feature that is not implemented in RoLL can be programmed for a specific learning problem with LISP functions, or, if the issue has been understood thoroughly and can be formulated for general purposes, RoLL can be enhanced further using LISP macros. One objective of this work was to provide a useful tool for making robot learning more efficient and for better understanding the challenges in this context.

One challenge of robot learning is how to select actions that give rise to interesting experiences — or how to implement a sophisticated problem generator. This question is anything but trivial. It doesn't only involve the question whether to choose an action that presumably results in the highest reward or one that returns information about the state space. How can the robot know that an action will result in experience it needs? The representation of raw experiences can be very different from the abstract feature language used for learning. It is this latter state description where the robot can detect if parts of the state space are unexplored, but for getting the experience, this high-level description must be translated into the representation of the raw experience. For example, the robot might want to know in which situations it drops things most likely. If it doesn't want to wait until it has broken every object in the kitchen, it should use objects made of plastic or some other robust material and try situations in which it is likely to drop things. But how can it provoke such situations? In order to know which activities are most promising for getting the experience it would need a model of when things are likely to be dropped, which is exactly what it is trying to learn.

Closely related to the issue of problem generation is the management of already acquired experiences. Usually robots don't act for longer periods of time. This is different in a realistic kitchen scenario, where they should operate for months or years. For continually adapting all models and routines to the environment, it would be impossible to collect experiences all the time. Instead, the robot should check from time to time if its control program is still adapted to its environment. The new experiences must be compared and weighed with respect to older ones, so that newer observations could be given more impact without discarding the old ones completely. A good knowledge and understanding also contributes to reasonable problem generation. Only when experiences can be understood on a high level, the robot can decide which observations must still be made.

Furthermore, the learning problems we have solved were integrated into the control program by an explicit call to `acquire-experiences` and `learn`. A better approach would be to declare learning problems, possibly provided with an indication as to their priority, and schedule experience acquisitions and learning processes automatically, so that the robot can use idle times for learning and active experience acquisition. This would of course require a good understanding of when the robot has time to spare and models of the time a learning process needs to run.

Finally, defining learning problems is still an art when it comes to deciding on a learning algorithm and its parameters. There are already some approaches for generating the

learning bias (see Section 6.7), thus enabling more automatic learning. A related question is the choice of appropriate abstractions, which is also discussed below. When the parameterization of learning problems could be fully automated, this would allow robots to create new learning problems when needed. This means, the programmer would be freed of the task to decide which parts of the program are to be learned. One would rather specify a simple programmed plan and let the robot itself decide which parts should be subject to optimization.

These issues have not yet been examined in great detail. One contribution of RoLL is to make these problems explicit. Another merit is that RoLL is an ideal tool for research in these areas, because it provides the basic learning mechanisms and comes with the complete LISP language to enhance and modify them. So there is a mutual dependency between RoLL and researchers in robot learning. The latter can profit from RoLL's capabilities to develop more sophisticated approaches and future developments of RoLL depend on their results.

6.6 Discussion

In this evaluation we corroborated the claims we made for RoLL. Unfortunately, we cannot compare our approach to other languages, because to our knowledge, there is no other programming language that incorporates the whole learning process including experience acquisition.

Besides, it is impossible to verify the value of such a language by empirical tests, because such tests could only consist in solving learning problems. However, on the one hand, a successful learning process wouldn't prove the value of RoLL, because the same learning problems can be solved without it as is being done in current robotic projects. On the other hand, successful learning depends on parameters to be chosen for the given problem, such as the learning system, its parameterization, the experience abstraction, and the raw experiences. Thus, a badly learned problem can be seen as proof that robot learning is difficult, but not as a flaw of RoLL.

The difficulty of showing the virtues of RoLL is an instance of the general problem of comparing programming languages. Every problem that can be solved in LISP, can in principle be solved in assembly language as well. So how can the advantages of a high-level programming language like LISP be proven? One way is to compare the run-time of benchmark problems. But there is no proof that the quality of the implementations in the two languages is comparable (we don't want to compare programmers' skills) and this criterion would probably favor assembly language, which is not what we would want to show. The most striking argument in favor of LISP in contrast to assembly language would clearly be the program code. This is why we provided a detailed example and pointed out the benefits of the RoLL concepts.

6.7 Related Work on Robot Learning

We have shown some examples of improving a robot's capabilities by learning. As this is not the main topic of this work, the presented learning problems are unpretentious. To demonstrate how learning methods can contribute to the success of robotic systems, we present some examples where learning contributed largely to the success of robotic systems.

Another facet of our evaluation was to show the flexibility of RoLL in using different ways of experience acquisition, switching learning systems, and comparing different parameterizations of learning problems. The goal of this work is not to solve any open questions on robot learning, but RoLL is certainly a convenient tool for developing and testing new methods for experience acquisition (especially problem generation), experience management, and learning algorithms. Therefore we will point to some methods that enhance learning and can be implemented in RoLL either in the form of a program for specific problems or for a general solution as an extension of RoLL.

Although RoLL emphasizes the combination of programming and learning, we think it appropriate to present some work on developmental robotics. This approach holds the opposite view of completely programming robots: it assumes that all behavior should be learned from the start like in small children. This field of research is interesting in the context of RoLL, because (1) RoLL provides an ideal programming environment for developmental learning, and (2) many topics mentioned before concerning experience acquisition or the chaining of learning problems have been studied in developmental robotics.

6.7.1 Case Studies

Learning on autonomous robots has been practiced strongly in the last years. Here we only present some examples of successful applications without trying to be exhaustive.

In the RoboCup domain complex tasks have been dealt with. Stone and Sutton (2001) learn a keep-away task in the simulation league by using SARSA-learning and linear tile coding, along with various adaptations such as predefined hand-coded skills and a reduction in the number of players. Work described by Takahashi and Asada (2001) focuses on stronger integration of control and perception, with a hierarchical learning approach, applied to the complex tasks in the middle-size league. Röfer (2005) employ evolutionary algorithms for developing an omnidirectional gait for the Sony AIBO robot. Gabel et al. (2006) and Lange and Riedmiller (2005) have employed learning algorithms both in the simulation and middle-size leagues to improve the robots' skills and for adaptive vision.

Another work on learning control for real robots is the one by Smart (2002), in which a robot improves its behavior through reinforcement learning in the real world learning parameters that would have to be tweaked by programmers otherwise. Crites and

Barto (1996) describe a reinforcement learning approach to controlling multiple elevators.

The most impressive robotic application of the last years has surely been Stanley, the winning car of the DARPA Grand Challenge (Thrun et al. 2006). In contrast to other teams, who tried to program every possible contingency, the strategy of Thrun et al. was to make the robot flexible enough so that it can handle all kinds of situations autonomously by learning from experience.

6.7.2 Methods

RoLL is the first approach to integrate full learning capabilities into a robot control language. When designing and using RoLL, one gets aware of how many intricate issues there are in the context of learning, some are presented by Blumberg (2002), where the learning process in animals is examined. Some of these problems have already been addressed in current research, like exploration strategies, others not so much, for example the management of experiences over longer periods of time. We now present some aspects of the learning process that have been explored more or less thoroughly, but aren't integrated as a general concept into RoLL. Because RoLL offers a complete programming environment, all of these techniques can be used. When such approaches are employed in several learning problems and a general concept has been recognized, RoLL can be extended by way of LISP macros to incorporate these ideas in a general way.

We deliberately omit references to learning algorithms, because the field is too wide. The learning system concept of RoLL is so general that it can handle all kinds of learning algorithms, as long as they rely on experiences that are accessible from within the program.

Exploration Strategies

The problem generator element in RoLL can only precompute situations that are to be explored. But usually the problem and its domain are unknown — that's why there is a learning problem. Thrun (1998a) gives a general survey of the differences in passive and active exploration. Although random generation of experiences has been used in many problems, the number of experiences can be reduced significantly when the state space is explored wisely.

In the field of reinforcement learning this problem is known as the trade-off between exploration and exploitation. The choice of the next action should help to explore the state space further. On the other hand, the already learned results make the robot's primary activity more efficient. An overview of this research branch is provided by Thrun (1992).

Cohn (1996) has shown in connection with neural network learning that by thorough experiment design not only the number of experiences is reduced, but that the performance

of the learning algorithm is enhanced.

Experience Abstraction

Another issue, on which the learning success heavily depends, is the choice and composition of the attributes used for learning, i.e. the feature language the experiences are translated to. Caruana and Freitag (1994) propose a greedy selection process of attributes for decision tree learning. Bonnlander (1996) compares different methods of choosing input values for neural network learning.

The choice of input variables gets harder for continuous domains. Herrera et al. (2006) describe a decision theoretic method for selecting a set of attributes with continuous values. A more complex task is the attribute selection when not only a subset of a given set of values is to be chosen, but when the possibility of operations between attributes is provided. This problem has been explored by Stulp, Pflüger, and Beetz (2006), who perform a greedy search in the space of given attributes and arithmetical combinations thereof. As the search space is extremely large, only abstractions for very simple problems can be found with this method.

Meta-learning

A very interesting field in the context of RoLL is the topic of meta-learning — automatically finding the bias for learning problems. On the one hand, RoLL offers an ideal tool for developing meta-learning methods, on the other hand, the integration of meta-learning facilities into RoLL would make the learning process more autonomous and enhance the learning performance. Vilalta and Drissi (2002) give a survey of the field.

Meta-learning is concerned with finding an optimal bias, i.e. the parameters guiding the performance of the learning system including the learning algorithm itself (Gordon and Desjardins 1995). Abraham and Nath (2000) describe a method for finding appropriate architectures for neural networks depending on the problem. They use a hybrid approach of genetic algorithms and simulated annealing search. Abraham (2003) reports on a method to choose the complete bias for neural network learning — the network structure, activation function, weight initialization function, and the learning function — automatically using evolutionary algorithms. Younger, Hochreiter, and Conwell (2001) use neural networks for the meta-learning process to find biases for neural networks.

More general than finding parameters for a certain learning paradigm is the choice of the learning algorithm from a wider set of learning algorithms. Pfahringer, Bensusan, and Giraud-Carrier (2000) describe an approach for classifying learning problems into classes and based thereon chose a learning algorithm. Instead of trying to find one algorithm for a learning problem, Brodley (1995) developed a procedure that combines the results of different learning systems to yield a hybrid classifier adapted to the specific problem.

Perturbations in the rewards can seriously affect reinforcement learning (RL) results. Anderson et al. (2006) supervise the RL process in a metacognitive loop and react to perturbations. This makes the learning process more stable and reliable in uncertain environments.

To our knowledge, there is no approach combining all method of exploration, extraction of useful features for the learning experience and meta-learning. One reason is certainly that there was no common framework for specifying, performing and testing learning problems. With RoLL these techniques and combinations of them could be explored to more depth and finally be integrated as language features.

Multiple Problem Learning

Most research in machine learning is focused on improving the performance of single learning problems. However, an agent performing sophisticated real-world tasks has to learn many different things: control routines, prediction models, optimization parameters, and plans. But learning many problems requires a lot of resources, because each problem needs experiences and time to learn. There have been some research efforts to explore ways in which learning problems can build on former learning results and how experiences can be re-used.

Mitchell (1990) applies explanation-based learning to compute plans into reactive rules. With a library of stimulus-response rules, the robot can act reactively to most situations. Only when there is no rule matching the situation, a plan is generated. This plan is then compiled to a reactive rule, which makes the robot more responsive without the need of planning.

Singh (1992) addresses the decomposition of reinforcement learning problems. Often RL is applied to big, monolithic problems with huge state spaces. The learning of such problems takes lots of experiences and time and the prospect of success is doubtful. This is why Singh explores how RL problems can be decomposed into elementary problems. Once these elementary tasks have been learned, their solutions are utilized for composite problems.

The most general research on performing multiple learning problems for constant improvement goes under the name of “lifelong learning” introduced by Thrun (1994) and Thrun and Mitchell (1993). The basic assumption is that it is necessary to solve learning problems building on other learned functions for a robot to evolve during longer execution periods like days or weeks. Thrun and Mitchell mainly address the problem of how to acquire general knowledge that is usable for several learning tasks so that only few problem-specific experiences are required. Two algorithms are presented for performing this knowledge transfer and also the problem of enabling robots to work in multiple environments is addressed.

6.7.3 Developmental Robotics

The field of developmental or epigenetic robotics is inspired by the learning capabilities of children. The basic idea is to build robots that learn their complete behavior by themselves (Weng 2004; Weng et al. 2001). So this is exactly the other end of the scale from a viewpoint of traditional robotics, where programming is the basic principle, enriched only by occasional learning. Beside the focus on robotic development, a lot of research is directed on the autonomous acquisition of language skills (Mukerjee and Sarkar 2006). The methods are often derived from findings in psychology.

When all the behavior is to be learned, not only one, but a whole set of learning problems has to be solved. This makes research in developmental learning especially interesting for RoLL (and vice versa) and runs along the lines of the approaches described in Section 6.7.2 (Bonarini et al. 2006). Other aspects of learning discussed in the previous sections are examined in the light of developmental learning: meta-learning (Valsalam, Bednar, and Miikkulainen 2006), knowledge transfer across several learning problems (Fei-Fei 2006), recognition of social behavior models (Crowley, Brdiczka, and Reignier 2006), and active experience acquisition (Oudeyer et al. 2005; Oudeyer, Kaplan, and Hafner 2007).

The behavior of self-developed robots does not exceed walking or simple communication tasks. No application in the scale of a household robot has been reported on in this field. This shows clearly that by learning alone — at least with the current state of the art — robots perform better with traditional approaches. On the other hand, purely programmed robots neither show the desired behavior and by far lack flexibility, especially in unknown situations and environments. RoLL proposes an approach in-between: easily-programmable, well-understood control strategies can be programmed, while environment-specific, flexible functions, models and routines are supplemented through learning.

Chapter 7

Conclusion

After giving a short summary, we provide a perspective of the overall research visions related to this work. We conclude by explaining the contributions of RoLL in this wider context.

7.1 Summary

This work presents a novel robot control language RoLL that allows to integrate learning capabilities smoothly into robot programs. The motivation of this work came from a machine learning perspective to develop a programming language supporting learning as proposed by Mitchell (2006) and from the need in autonomous robot research to develop constantly improving robots.

Developing a declarative language that includes the possibility of automatic experience acquisition demands a uniform formal framework. We use hierarchical hybrid automata for modeling the program execution in the environment and define the experiences based on this model. An experience is a summary of the robot's internal state observed during an episode, which is specified according to the execution model. This view on experiences is maintained for abstract experiences. Besides, not only experiences, but the whole learning process can be described by hybrid automata when the models to be learned are described in the hybrid automaton framework.

On the basis of the architecture for learning agents and the hybrid automaton view on learning, we have presented the language RoLL, which contains learning constructs as an extension to the existing robot control language RPL. The constructs comprise the complete learning process with experience acquisition and abstraction, learning with arbitrary experience-based algorithms and integrating the result into the program. This procedure is a very general view on learning so that different learning paradigms can be used. It allows active and passive experience acquisition as well as online and offline learning.

The experience acquisition is independent of the primary control program and very powerful with respect to the episode detection and data that can be observed. The monitoring process has access to all internal information of the robot like its program state, local variables and its belief about the external state. The integration of the learning results is performed automatically based on the abstraction used for learning and the specified learning problem class.

The field of machine learning and especially methods to use learning on autonomous robots is evolving very quickly. Open research questions include intelligent problem generation and meta-learning. To allow a broad range of learning procedures to be used with RoLL, some aspects of the learning process are explicitly open for modular extension. This is the case for learning systems, experience storage systems and learning problem types. Developments in learning along other lines, e.g. problem generation, can be programmed with the underlying LISP language. To adapt it permanently to the latest findings in the field, RoLL can be extended using LISP macros.

We have demonstrated the usability and elegance of RoLL by presenting an example for a complete learning problem. We have further shown empirical results of learning problems solved with RoLL. In this context we pointed out some advantages of using RoLL compared with the manual method like repeatability, automatic evaluation, and using different abstractions.

7.2 Related Visions for Robotics and AI

After the success of early AI programs, the field was faced with ambitious expectations, which is mirrored in the great number of science fiction stories, where intelligent computers, robots and cars support humans in their daily activities (or take over the world ...). Although there are some successes like speech recognition, navigation systems and internet search engines, most of these visions are far beyond the current state of the art.

One reason is that the shift from coarse simulations and games to technical devices operating in the real world is much more intricate than everybody thought. But this is not the only explanation. Marvin Minsky argues in a 2001 talk that the field has drifted to very specialized research areas like neural networks or genetic algorithms, which are unquestionably interesting to be explored, but should not replace the work on other interesting questions.

> So the question is why we didn't get HAL in 2001? I think the answer is I believe we could have. I once went to an international conference on neural nets. There were 40 thousand registrants. I don't know how many go to an international conference on genetic algorithms or genetic programs, but there are many thousands of people working on that. Tens of thousands of people

> try to make slightly better rule-based systems. That's another very useful representation. But very few try to make a system that will in some very smart ways make new rules on the basis of experience. That's the obvious thing to do.
>
> Minsky (2001)

Besides, he emphasizes the necessity for a robot to have models of itself and to improve with experience.

> Making and using models of yourself, like revealing the course of your ability to solve a problem. You might say well, maybe I need to take a course in this. Or if I don't get better at this, my friends will hate me. Really if I did get better at this, I'd just be a nerd of some sort and my friends would hate me even more. And so on. Hearing conversations in your mind. Making new goals. Adjusting your level of wakefulness, whatever that means.
>
> Minsky (2001)

This attitude is slowly gaining ground in the research directions of influential decision makers. Under the name of "Cognitive Systems" there are already a few projects striving to build integrated technical systems that use the findings of the specialized research, which has been performed over the last two decades, and bring together findings from computer science, mechanical and electrical engineering, as well as results from psychology and neuroscience. The goal is to have "systems that know what they're doing" (Brachman 2002), or more specifically that learn, reason, react to unknown situations, take advice and explain their doings.

The Defense Advanced Research Projects Agency's Information Processing Technology Office (DARPA ipto) supports several projects in this direction. We only mention the ones that explicitly demand research on learning.

The Project "Personalized Assistant that Learns (PAL)" aims at supporting humans in everyday environments. Such systems must be able to reason, accept user commands in a natural way, explain their doings, learn from and reflect on their experience and respond to surprise. A main issue is the automatic improvement of the basic functionality by acquiring experience and deciding what to learn. This mission is awarded to to SRI's CALO (Cognitive Assistant that Learns and Organizes) project[1] and the RADAR project at CMU[2]. The functionality to be supported by CALO includes meeting preparations, assistance in creating documents, and managing information (Myers 2006).

A very important area of robotic applications are ground vehicles. In the DARPA ipto project "Learning Applied to Ground Robots (LAGR)" the performance of robotic

[1] `http://calosystem.org`
[2] `http://www.radar.cs.cmu.edu/`

vehicles is enhanced by constant learning from own experience and by imitating human behavior. The "Integrated Learning (IL)" project addresses the issue of learning with scarce experience. The idea is to use all available sources of information, not just the experience at hand to improve learning results. Ideally, only one example suffices to improve the behavior.

Other funding agencies are aware of the growing need to develop cognitive applications. The Deutsche Forschungsgemeinschaft (DFG) has accepted the cluster of excellence Cognition for Technical Systems (CoTeSys)[3] to be supported over the next years (Buss, Beetz, and Wollherr 2007). This cluster brings together researchers from different fields and different institutions to develop fully functioning cognitive technical systems in different application domains.

To reach these ambitious goals, learning must be an integral part of the systems. In order to respond to previously unknown situations such situations must be detected in the first place. This is possible when the system has models telling it what a normal situation would be and compare them to the circumstances at hand and it needs ways to monitor its behavior constantly. The models must be learned and updated continually. The response to the changed situation can be achieved by integrating new observations into the set of previous ones. Users expect an intelligent system to adapt to their preferences and not to make the same mistake twice.

7.3 RoLL's Contributions to More Sophisticated Autonomous Robots

After having related the challenges of and work on cognitive systems, we point out the issues where RoLL can contribute to achieving these ambitious objectives.

One feature of cognitive systems is that they are to serve as reliable personal assistants over long time periods while adapting to the needs of the user. As the preferences of users and their tasks and responsibilities change over time, it is out of the question to learn only at the beginning and then stop the learning process. Continual learning involves many open questions:

- How should learning problems be scheduled to build upon each other?
- How does a robot recognize a stream of data as an experience?
- How can experiences be used for different learning problems?
- What are interesting experiences?
- How are the learning steps integrated into a program?

[3] `http://www.cotesys.org`

- How can the learning be performed automatically? This involves the choice of the learning algorithm and bias as well as the generation of an appropriate abstraction.
- How must a program be constructed so that the learning results can be integrated?
- How can the program find out which problems it must learn?

None of these issues has been solved completely, some even haven't been examined at all. Although RoLL doesn't give answers to all of these questions, it provides solutions to some of them and methods for investigating these issues to greater depth. The concept of experiences is well-defined and understood in RoLL. Experiences can be specified and acquired automatically. The basis of more sophisticated experience management is provided by an interface to a database for storing experiences. Using data mining methods, experiences can be analyzed and understood by the learner. This allows a more economic use of experiences and goal-directed methods for observing them actively.

Furthermore, RoLL enables a smooth interaction between programming and learning. Routines or functions are replaced by learned ones without the rest of the program being changed. And the experience acquisition is integrated in a way that doesn't disturb normal program development, but can be added when needed. With the combination of declarative specifications and automatic execution, RoLL is a first step on the way of automating the learning process completely. With enhancements from meta-learning the programmer doesn't have to be a specialist in learning algorithms and can be freed more and more from tedious parameter tweaking.

RoLL can also be seen as a tool for developing better learning algorithms and to examine general properties of known learning algorithms, which is not only important for robotic applications, but for machine learning in general. Learning problems can be repeated and tested with different parameterizations so that learning algorithms can be evaluated comfortably. Other open questions of learning can be explored similarly, like how to acquire experience actively, how to use results of one learning problem in another or how to perform learning successfully with only few experiences. As RoLL comes with the whole functionality of LISP, new methods can be added easily and be combined with and tested against known methods.

Beside all this, the most fundamental contribution of RoLL to cognitive systems is its implementation of a first programming language to include full learning capabilities. This breaks the traditional distinction between programming and learning, which is present in almost all current systems. Only when learning is seen as an integral, indispensable component of the control system, cognitive systems have a chance of success.

Appendix A

RPL Language Overview

This chapter gives a short introduction to RPL and a reference to RPL language constructs used in this work. For a better understanding of RPL and a more detailed description of the language see (McDermott 1993).

A.1 Basic Concepts

RPL is defined as a macro extension of LISP. Its basic control concept are tasks that are managed in a control loop similar to the way operating systems work. This means that RPL makes sure that each task gets a chance to run, that they work in parallel and interact without blocking each other. The unique feature provided by RPL is that the hierarchy of active *tasks*, their status and interaction can be accessed by the program at run-time. As the task descriptions also contain all local variable bindings, one task can examine the variable state of another. The status of a task can be categorized with these values: *created, enabled, active, done, failed, evaporated, current, passed*. Figure 3.4 on page 33 shows the code tree for a plan and how its parts can be addressed (see also the description of the `path-sub` construct).

Another distinctive feature of RPL is the concept of *failures*. The handling of failures is implicit in many control structures. For example, when there are several ways to reach a goal, the robot can pursue all of them at once and stop when one has succeeded. If one option fails, there is still a chance that one of the other possibilities achieve to produce the desired effect. In other cases, a higher-level control structure fails, when only one of its subplans fail, propagating the error higher up in the hierarchy in the hope that higher-level plans can deal with the failure appropriately. This is why RPL has several constructs for indicating parallel execution (`par`, `pursue`, `try-all`), each handling failure and success conditions differently.

Furthermore, RPL uses the concept of *fluents* to handle variables that change their

value continuously. These are mostly variables associated with values in the real world like the robot's current position. Several control structures explicitly react to changes in fluents without having to check on variable values explicitly all the time. Such commands include `whenever` and `wait-for`. Fluents are also used for indicating the state of tasks. Each task is associated with fluents for when it begins and ends. These fluents are set from false to true when the respective event happens and reset to false shortly afterwards. Even more interesting than single fluents is the possibility to build networks of fluents. 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 (see Figure 3.3 on page 30).

A.2 Relation to Lisp

RPL is a macro extension of LISP and therefore supports all LISP commands. However, the working of LISP is not captured in the internal RPL representation. When RPL encounters a command it doesn't know it assumes it is pure LISP code and interprets it as such. The interpretation of LISP code is a black box to RPL. Thus, inside LISP code no RPL commands are allowed. For example

```
(progn
  (setf c (+ a b))
  (wait-for f))
```

is illegal unless there exists a LISP function with the name `wait-for`, the RPL `wait-for` cannot be used here, because the LISP construct `progn` is not known by RPL. Therefore the RPL context is left and control is passed to LISP.

Because programmers are used to certain LISP constructs, RPL offers commands with the same name and identical behavior, but in the context of RPL execution. These are `let`, `let*`, `if`, `when`, `cond`, `unless`, and `loop`[1]. It depends on the context if the LISP or RPL commands are used. For example in

```
1 (def-rpl-method execute ((ctr controller))
2   (let ( (a (create-fluent "x" nil))
3          (b (1+ (if global-c global-x global-y))) )
4     (if (> b 2)
5       (set-value a t)
6       (setf b (1+ b)))))
```

the `let` in line 2 and the `if` in line 4 are both executed in RPL mode, because the RPL context is never left on the way. In contrast, the `if` in line 3 is executed as LISP code.

[1] In the original RPL only `let`, `let*`, `if`, `loop` are defined, the others have been added for RoLL.

A.3 RPL Language Constructs

We now list some RPL commands used in this work ordered by functional aspects. The descriptions are copied from (McDermott 1993) and (McDermott 1992), some annotations being added.

Fluents

Basic Operations:

`(create-fluent` *`pat v`*`)` Create a new fluent with initial value *`v`*. The *`pat`* is for mnemonic purposes in printing the fluent.

`(changed-fluent` *`f compare-fcn`*`)` A fluent that is momentarily set to `true` whenever the value of fluent *`f`* changes. The *`compare-fcn`* is an equality test that is used to tell whether the value has changed.

Constructs Reacting to Fluent Changes:

`(whenever` *`p -body-`*`)` Do *`-body-`* once every time *`p`* goes from false to `true` (including once if it's true when the `whenever` starts).

`(wait-for` *`p`*`)` *`p`* should evaluate to a fluent. Wait until *`p`* becomes true.

`(wait-time` *`t`*`)`[2] Wait for *`t`* seconds.

Control Flow

Loops:

`(loop` *`-body-`*`)` Do actions in *`-body-`* in sequence, then repeat. The body may contain tests of the form `while` *`e`* or `until` *`e`*. If *`e`* evaluates to the right Boolean value, the loop exits.

`(n-times` *`n -body-`*`)` Like `loop`, but quits after *`n`* iterations if no other test has caused it to exit.

Parallel Execution:

`(par` a_1 a_2 `...`a_n`)` Do actions in parallel. Succeed if they all succeed; fail if one fails.

`(pursue` a_1 a_2 `...`a_n`)` Do actions in parallel. Succeed if one succeeds (the others evaporate), fail if one fails.

`(try-all` a_1 a_2 `...`a_n`)` Do actions in parallel. Succeed if one succeeds (the others evaporate), fail if all fail.

[2] Although `wait-time` doesn't involve fluents, it is listed here because of its similarity to `wait-for`.

`(with-policy p -body-)` Do *-body-*, but also do *p*, never allowing *-body-* to run unless *p* is blocked. Fail if either *-body-* or *p* fails, succeed if *-body-* succeeds.

Others:

`(no-op)` Succeed. That is, do nothing.

`(seq` a_1 a_2 `...`a_n`)` Do each action in order.

`(evap-protect a b)` Normally means the same as `(seq a b)`, but if *a* should evaporate because the plan containing it gets swapped out, then *b* gets executed anyway.

Task Inspection

Task Surveillance:

`(begin-task t)` A Boolean fluent that gets value `true` when task *t* becomes active.

`(end-task t)` A Boolean fluent that gets value `true` when task *t* becomes *finished*, *evaporated*, or *failed*.

Task Addressing:

`(:tag l a)` This is not an executable form. It's equivalent to *a*, but if it occurs in a top-level plan or procedure body, then *l* (a symbol) will be bound to the task for *a* at the highest level in the plan where there is a single task for *a*, i.e., the highest non-iterative context, or the whole plan.

`(path-sub -name-prefixes- task)` The syntactic sub-sub-...-task of *task* arrived at by using the elements of *-name-prefixes-* in reverse order. For example, for addressing the task γ from the top-level task in Figure 3.4 on page 33, let's assume the top-level task is bound to the variable `tk`, use `(path-sub ((if-arm false) (branch 2)) tk)`.

Appendix B

Hybrid Automaton Definitions

Hybrid Automaton
A hybrid automaton (HA) is defined by a tuple $\mathscr{H} = \langle V,\ M,\ flow,\ T,\ act,\ cond \rangle$ with

- a finite set of variables V.
- a set of modes M
- a flow function assigning each mode a function that describes the continuous change in the mode $flow : M \to (\Sigma \to \Sigma)$
- a set of transitions $T \subseteq M \times M$ between modes;
- an action function $act : T \times \Sigma \to \Sigma$ assigning each transition a change in the data state;
- jump conditions *cond* assigning a predicate to each transition.

Hierarchical Hybrid Automaton
A hierarchical hybrid automaton (HHA) is a tuple $\mathscr{H} = \langle V,\ M,\ flow,\ T,\ act,\ cond \rangle$ with

- a finite set of variables V.
- a flow function $flow : \Sigma \to \Sigma$ describing the continuous change in the set of variables during the execution of $\mathscr{H}$.
- a set of modes or subautomata M. The variables of the parent automaton are also variables of the subautomata: $\forall \mathscr{M} \in M : V_{\mathscr{H}} \subseteq V_{\mathscr{M}}$. Unless M is the empty set, $\mathscr{S}_{\mathscr{H}} \in \mathscr{M}$ is the starting mode, $\mathscr{T}_{\mathscr{H}}$ the terminating mode.
- a set of transitions $T \subseteq M \times M$ between modes;
- an action function $act : T \times \Sigma \to \Sigma$ assigning each transition a change in the data state;
- jump conditions *cond* assigning a predicate to each transition.

Probabilistic Hierarchical Hybrid Automaton
A probabilistic hierarchical hybrid automaton (PHHA) is specified by a tuple $\mathscr{H} = \langle V,\ M,\ flow,\ T,\ act,\ prob \rangle$ with

- a finite set of variables V.
- a flow function $flow : \Sigma \to \Sigma$ describing the continuous change in the set of variables during the execution of $\mathscr{H}$.
- a set of modes or subautomata M. The variables of the parent automaton are also variables of the subautomata: $\forall \mathscr{M} \in M : V_{\mathscr{H}} \subseteq V_{\mathscr{M}}$.
- a set of transitions $T \subseteq M \times M$ between modes;
- an action function $act : T \times \Sigma \to \Sigma$ assigning each transition a change in the data state;
- a probability function $prob : T \times \Sigma \to [0,1]$ assigning to each transition a probability table, which means that a transition only occurs with the probability given in the probability table according to the current state. For a given jump condition σ from a given mode m, the probabilities of different outcomes must sum up to 1:

$$\sum_{m'} prob((m,m'),\sigma) = 1.$$

Data State
A *data state* of a set of variables V is a function $\sigma : V \to D$ assigning each variable $v_i \in V$ a value from some basic set D, e.g. the real numbers. The set of all data states is called Σ.

Program State
A *program state* of a HHA $\mathscr{H}$ is a function $\gamma : M \cup \mathscr{H} \to \{0,1\}$ assigning a value 0 or 1 to each mode and the automaton itself. Stated differently, γ is the characteristic function of the set of active automata in $\mathscr{H}$. The set of all program states is Γ.

Automaton State
An *automaton state* θ of a HA $\mathscr{H}$ is a tuple of a program state and a data state over the variables $V_{\mathscr{H}}$ of the automaton: $\theta \in \Gamma \times \Sigma$. The set of all automaton states is called Θ.

Interval
An *interval* is a connected portion of the real line. If the endpoints a and b are finite and are included, the interval is called *closed* and is denoted [a,b].

A non-empty subset X of $\mathbf{R}$ is an interval iff, for all $a,b \in X$ and $c \in \mathbf{R}, a \leq c \leq b$ implies $c \in X$.

Automaton Activation Period
The time during which an automaton is active is called *automaton activation period*. It is represented by a closed interval $[begin, end]$, whose first point is called *begin* and whose last point is called *end*.

Episode
An *episode* $e^I_{\mathscr{H}}$ of an automaton $\mathscr{H}$ is a function mapping the real values of an interval I to program states of $\mathscr{H}$.

State Projection
Given a set of variables V and a data state σ, a *projection* of a set $W \subseteq V$ on σ is $\sigma_W : W \to D$, $\sigma_W(v) = \sigma(v), v \in W$.

Automaton Transformation
An automaton transformation is a mapping from one HA $\mathscr{H}$ to HA $\mathscr{I}$ using any of the three following operations:

- abstract: $\mathscr{H} \to flow_{\mathscr{H}}$
- expand: $\mathscr{H} \to M_{\mathscr{H}}$
- restrict: $\mathscr{H} \to N$, where $N \subset M_{\mathscr{H}}$

Appendix C

RoLL Reference

Table 5.1 on page 67 shows a summary of the RoLL constructs. This chapter provides a detailed reference manual on their usage.

For describing the language constructs we use a BNF-like notation, which is explained in Table C.1. Each construct is specified formally using this notation. Additionally, the purpose of the construct and a short description are provided. A complete example of a learning problem definition using these expressions is presented in Chapter 6.2 on page 104. The usage of the constructs for extending RoLL's functionality is shown in Appendix D on page 158.

Table C.1 Explanation of the notation for RoLL constructs.

`define-...`	Constructs and their syntactical components of RoLL are written in `typewriter` font.
optional	An optional expression is illustrated in gray.
default	Default values are indicated by underlining.
⟨lisp expression⟩	A construct from LISP or RPL giving the type or a short specification;
⟨name $_{\text{lisp type}}$⟩	A construct from LISP or RPL giving an explanation with a significant name and providing the desired LISP type.
⟨*expression*⟩	An expression that is defined later or has been defined on the same page.
⟨*expression*^{page}⟩	An expression that has been defined on the page indicated.
a → b	'a' may be of the form 'b'.
a | b	Alternative construct, either 'a' or 'b'.
a^+	→ a | aa^+.
a^*	either nothing or a^+;

C.1 Experience Data

The addressing of data from experiences is solved in a uniform way in RoLL. Before explaining the single constructs, we define the access to experience data as a reference in later definitions.

⟨*automaton data*⟩ → `:begin` | `:end` | `:interval`
| (⟨*event*⟩ ⟨*automaton*⟩ ⟨*occurrence specification*⟩)
| (`:interval` ⟨*automaton*⟩ ⟨*occurrence specification*⟩
⟨*instance specification*⟩)

⟨*event*⟩ → `:begin` | `:end`

⟨*automaton*⟩ → ⟨automaton name $_{\text{symbol}}$⟩

⟨*occurrence specification*⟩ → `:first-occurrence` | `:last-occurrence`
| `:only-occurrence` | `:all-occurrences`
| ⟨nth occurrence $_{\text{integer}}$⟩

⟨*instance specification*⟩ → `:first-instance` | `:last-instance`
| `:all-instances` | ⟨nth instance $_{\text{integer}}$⟩

Purpose: Access data of experiences.

Description: Access the *begin, end* or *interval* slots of an automaton. If the automaton is anonymous, it needn't be named explicitly. If the automaton is named in the definition, it must be addressed by its name. For events, an optional occurrence specification may be given, for intervals the instance can be specified additionally, see explanation on page 69.

⟨*automaton var data*⟩ → (`:var` ⟨variable $_{\text{symbol}}$⟩ ⟨*automaton data*⟩)

Purpose: Access detailed data of experience structure.

Description: Access single variables from the *begin, end* or *interval* slots of an experience automaton.

⟨*crude automaton data*⟩ → `:begin` | `:end` | `:interval`
| (⟨*event*⟩ ⟨*automaton*⟩) | (`:interval` ⟨*automaton*⟩)

Purpose: Access data of experience structure in situations where occurrence and interval specifications make no sense.

Description: Like automaton data, but no occurrence or interval specifications allowed.

C.2 Raw Experiences

(`define-raw-experience` ⟨experience name $_{\text{symbol}}$⟩
`:specification` ⟨*experience automaton*⟩
`:experience-handling` (⟨*handling instruction*⟩$^{+}$))

Purpose: Declaratively define a raw experience.

Description: Define an experience automaton including the data to be acquired during an episode. Experience handling describes how to treat the data in case of special occurrences like failures while the automaton is running.

C.2.1 Experience Automaton

⟨*experience automaton*⟩ → `(`⟨*automaton identifier*⟩
`:rpl` ⟨*task*⟩
`:invariant` ⟨fluent⟩
`:maximum-duration` ⟨number⟩
`:events ((`⟨fluent⟩ ⟨event name $_{\text{symbol}}$⟩$^{+}$`)`$^{+}$`)`
`:interval` ⟨*recording data*⟩
`:begin` ⟨*recording data*⟩
`:end` ⟨*recording data*⟩
`:children (`⟨*experience automaton*⟩$^{+}$`)`
⟨*automaton identifier*⟩ → ⟨dummy name $_{\text{keyword symbol}}$⟩
⟨automaton name $_{\text{non-keyword symbol}}$⟩

Purpose: Describe an experience automaton.

Description: The automaton defines the episodes to be observed either by an invariant (the automaton being active when the given fluent is not `nil`) or an RPL task specification. The data is attributed to the begin and end events or the complete interval of the automaton execution. Maximum duration and the specification of unwanted events by fluents are the available options for handling failures. With the `children` slot, a hierarchy of automata can be defined.

Episode Specification

⟨*task*⟩ → `(:goal-execution` ⟨goal $_{\text{symbol}}$⟩`)`
| `(:routine-execution` ⟨routine $_{\text{symbol}}$⟩`)`
| `(:goal-subtask` ⟨*task*⟩`)`
| `(:achieve-subtask` ⟨*task*⟩`)`
| `(:sub` ⟨subtask path⟩ ⟨*task*⟩`)`
| `(:tagged` ⟨tag name $_{\text{symbol}}$⟩`)`
| `(:recover` ⟨*failure task*⟩`)`
| `(:monitor` ⟨*failure task*⟩`)`
| `(:perform` ⟨*failure task*⟩`)`
| `(:try` ⟨*failure task*⟩`)`
| `:parent`
| ⟨supertask $_{\text{symbol}}$⟩
⟨*failure task*⟩ → `(:failure-handling` ⟨subtask path⟩ ⟨*task*⟩`)`
| `(:failure-handling-default` ⟨*task*⟩`)`

Purpose: Describing the program part to be observed.

Description: There are several possibilities: (1) reacting to a routine or goal execution, (2) addressing a subtask of another task specified in the same way, (3) a task defined by a parent automaton (either the direct parent or any other addressed by its name), if such an automaton exists. For defining explicit paths from one subtask to another the notation of RPL is used as illustrated by Figure 3.4 on page 33 and explained in the RPL manual (McDermott 1993).
When a task is defined by `with-failure-handling`, it must be addressed as a `failure-handling` task definition (`failure-handling-default` does the same, but assumes a certain program structure). The subtasks are then addressed by `recover`, `monitor`, `perform`, and `try` (summarizing monitor and perform).

Data Specification

⟨*recording data*⟩ → `(`⟨*variable definition*⟩$^+$`)`
⟨*variable definition*⟩ → `(`⟨variable $_{\text{symbol}}$⟩ ⟨*value description*⟩`)`
⟨*value description*⟩ → ⟨symbol⟩
| `(:internal-value` ⟨*local variable*⟩ ⟨*task*⟩ ⟨*search hint*⟩`)`
| `(`⟨operator $_{\text{symbol}}$⟩ ⟨*value description*⟩$^+$`)`
⟨*local variable*⟩ → ⟨string⟩ | ⟨symbol⟩
⟨*search hint*⟩ → `:exact` | `:prefix`

Purpose: Specifying the data to be observed.

Description: Each data slot is given a variable name. The values bound to these names are either drawn from a LISP expression (either symbol or LISP function) or from a local variable inside a task. Both sources can be combined by arbitrary LISP expressions.

C.2.2 Event Handling

⟨*handling instruction*⟩ → `(`⟨*handling condition*⟩ ⟨*handling reaction*⟩`)`

Purpose: Defining how to prepare the data for further abstraction.

Description: Depending on predefined or user-defined events that have occurred during execution, the data can be manipulated or deleted. The syntax is oriented at the LISP `cond` construct.

⟨*handling condition*⟩ → T
| (:event ⟨*monitored event*⟩)
| (:available ⟨*automaton data*⟩)
| (:available ⟨*automaton var data*⟩)
| (:all-available ⟨automaton name $_{\text{symbol}}$⟩)
| :all-available
| (⟨operator $_{\text{symbol}}$⟩ ⟨*handling condition*⟩$^{+}$)
⟨*monitored event*⟩ → ⟨*event id*⟩ | (⟨*event id*⟩ ⟨automaton name $_{\text{symbol}}$⟩)
⟨*event id*⟩ → :timeout | :success | :evap | :fail | :done | :abort
| ⟨event name $_{\text{symbol}}$⟩

Purpose: Describing a handling condition of the experience.

Description: There are two ways to differentiate situations: (1) based on the events that were monitored during the automaton activity, and (2) the data available in the experience. A combination of both with arbitrary LISP operations or functions is possible. Monitored events are the RPL status of tasks when the automaton ends and events defined by the user in the automaton definition.
If a condition is defined as an event without providing an automaton name, the condition is fulfilled when any automaton in the hierarchy has produced such an event. When an automaton name is given, only the event occurrence in this automaton can fulfill the condition.

⟨*handling reaction*⟩ → :discard | :store | ⟨*data manipulation*⟩$^{+}$:store
⟨*data manipulation*⟩ → ⟨*data replacement*⟩ | ⟨*conditional replacement*⟩
⟨*data replacement*⟩ → (set-data ⟨*automaton data*⟩ ⟨LISP expression⟩)
| (add-data ⟨*automaton data*⟩ ⟨LISP expression⟩)
| (replace-data ⟨*automaton var data*⟩ ⟨LISP expression⟩)
| (replace-data ⟨*automaton data*⟩ ⟨LISP expression⟩)
| (erase-data ⟨*automaton data*⟩)
⟨*conditional replacement*⟩ → (⟨*cond*⟩ ⟨*handling condition*⟩ ⟨*data replacement*⟩$^{+}$)
⟨*cond*⟩ → when | unless | if

Purpose: Specify the reaction to a condition.

Description: The data can be discarded or stored completely, or it can be manipulated before being stored by deleting data, or adding to or replacing data by arbitrary LISP expressions. The manipulation can be guided by conditions described in the same way as the handling conditions on the top level.

C.3 Problem Generation

C.3.1 Defining Problems

```
(problem-parameters
  :parameters (⟨parameter⟩+)
  :relation ⟨relation⟩)
```

Purpose: Generate a list of problems.

Description: Problems are a vector of values attributed to some variables. With the construct `problem-parameters` a list of such problems is generated according to the specification.

⟨*parameter*⟩ → `(`⟨*variables*⟩ ⟨*parameter source*⟩`)`
⟨*variables*⟩ → ⟨variable $_{\text{symbol}}$⟩ | `(`⟨variable $_{\text{symbol}}$⟩+`)`
⟨*parameter source*⟩ → `:random :min` ⟨*min value*⟩ `:max` ⟨value $_{\text{number}}$⟩
`:samples` ⟨quantity $_{\text{integer}}$⟩
| `:cover :min` ⟨*min value*⟩ `:max` ⟨value $_{\text{number}}$⟩
`:interval` ⟨value $_{\text{number}}$⟩
| `:predefined` ⟨value $_{\text{T}}$⟩+
⟨*min value*⟩ → `0.0` | ⟨value $_{\text{number}}$⟩

Purpose: Define how values are generated.

Description: For each variable define a list of values, which can either be random, covering a range of values systematically or by giving an explicit list.

⟨*relation*⟩ → `(`⟨*relation keyword*⟩ ⟨*relation argument*⟩+`)`
⟨*relation keyword*⟩ → `:dep` | `:indep` | `:indep-min` | `:indep-max`
⟨*relation argument*⟩ → ⟨variable $_{\text{symbol}}$⟩ | ⟨*relation*⟩

Purpose: Combine the individual variable value lists into problems.

Description: Every problem consists of one value for each variable. The individual lists can either be combined with a cross product (`:dep`) or by taking one element of each list (`:indep`). `:indep` and `:indep-max` are synonymous.

C.3.2 Using Generated Problems

```
(with-problem-parameters (:parameters (⟨parameter⟩+)
                          :relation ⟨relation⟩)
  ⟨body RPL code⟩)
```

Purpose: Use a parameter list inside RPL code.

Description: Runs the body code in a loop, binding the variables defined in the parameters slot to one problem per loop run. Using the variables in the body, the code is executed with different parameterizations.

C.3.3 Examples

The syntax and working of the problem generator can best be explained by examples. The RoLL specification of the problem generator contains two parts: (1) the parameters and a description of what kind of values they should each adopt in subsequent program runs and (2) the relation between the parameters. The default relation between parameters is dependency.

Here is the first example of a problem generator specification:

```
(problem-parameters
 :parameters ((x :random :min 1.0 :max 2.0 :samples 2)
              (y :predefined '(4 5 6)))
 :relation (:dep x y))
```

With this declaration six pairs of parameter values are generated in total. For the parameter x two random values in the range between 1.0 and 2.0 are generated, for example 1.7654 and 1.286. The three possible values for parameter y are given in the specification. The two value lists for both parameters are then combined according to the relation specification. In this case the parameters are defined to be dependent, which corresponds to the cross product of the two lists, so that the resulting list of values would be `((1.8595252 6)(1.8595252 5)(1.8595252 4)(1.6935984 6)(1.6935984 5)(1.6935984 4))`. In each run of the acquisition program, one pair is used, so that in the first run, parameter x takes the value 1.7654 and y is bound to the value 4. In this example the relation specification is redundant, because the dependency of both values is the default.

In the following example we see how the number of problems is determined automatically. It is almost identical to the previous one.

```
(problem-parameters
 :parameters ((x :random :min 1.0 :max 2.0)
              (y :predefined '(4 5 6)))
 :relation (:indep x y))
```

Here the two parameters are declared to be independent and for parameter x the number of needed samples is not given. The overall number of problems is determined by parameter y and the number of needed samples for x is set automatically to three. A possible result of this specification is `((1.2603241 4)(1.228759 5)(1.1549773 6))`.

Now what happens if we declare x and y to be independent, but additionally specify that we want only two different values for x as we have done in the first example? The overall number of problems could be two, because for parameter x only two values are generated, but then we have to omit one of the values for y. Contrariwise, if we want all the possible values of y to be used, one of the values of x must be utilized twice. This ambiguity has to be resolved by the programmer. The relation specification `(:indep-min x y)` produces the first variant, for example `((1.6880503 4)(1.7453804 5))`, whereas with

the declaration `(:indep-max x y)` a list of three problems is generated: `((1.3849926 4)(1.7848289 5)(1.3849926 6))`.

If two parameters are to take the same values, the declaration can be abbreviated. For example, if a parameter `u` is to take the same possible values as `y`, we could declare `((y u) :predefined '(4 5 6))`. In this case with predefined values we could as well have copied the declaration for `y` and used it for `u`. If the values are determined randomly, however, the two declarations have different results.

```
(problem-parameters
 :parameters (((x y) :random :min 1.0 :max 2.0 :samples 2)))
```

This specification produces two random values, which are both used for parameters `x` and `y`, so that the resulting problem list might look like this: `((1.7241 1.7241)(1.7241 1.8092)(1.8092 1.7241)(1.8092 1.8092))`. In contrast, the following specification generates two separate lists of random values, which are then combined. The result could be `((1.3703 1.3379)(1.3703 1.8953)(1.8114 1.3379)(1.8114 1.8953))`.

```
(problem-parameters
 :parameters ((x :random :min 1.0 :max 2.0 :samples 2)
              (y :random :min 1.0 :max 2.0 :samples 2)))
```

Finally, we present a more complex example with different relations between the variables. We also see how a range of discretized values is covered completely.

```
(problem-parameters
 :parameters ((x :random :min 1.0 :max 2.0 :samples 2)
              (y :predefined '(4 5 6))
              (u :cover :min 0.0 :max 2.0 :interval 0.6)
              (w :cover :max 2.0 :interval -0.6))
 :relation (:indep-max (:dep x u) (:indep-min y w)))
```

The value specification of parameters `x` and `y` is the same as in the examples before, the number of possible values for `x` is restricted to two. The additional parameters `u` and `w` cover a range between 0.0 and 2.0 (the specification `:min 0.0` is redundant, as 0.0 is the default minimum value). The discretization step is 0.6 for both values, but the discretization of `u` starts at 0.0, which gives the values 0.0, 0.6, 1.2, and 1.8. In contrast, the values for `w` are 2.0, 1.4, 0.8, and 0.2.

Now let's have a closer look at the dependencies between the variables. Parameters `x` and `u` depend upon each other. As `x` provides two values and `u` four values, we get a total of eight problems for the combination. `y` and `w` are declared to be independent and the lower possible number of values is to be used. This means that three problems are proposed for the combination of `y` and `w`, one of the values of `w` being discarded. The overall combination of the sublists for `x`/`u` and `y`/`w` are to be combined as independent lists using the maximum number of problems that can be generated. The result is a list of eight

values (the ordering of the values is x, u, y, w as specified by the dependency relation):

```
((1.4037962 1.8 4 2.0) (1.4037962 1.2 5 1.4)
 (1.4037962 0.6 6 0.8) (1.4037962 0.0 4 2.0)
 (1.4152595 1.8 5 1.4) (1.4152595 1.2 6 0.8)
 (1.4152595 0.6 4 2.0) (1.4152595 0.0 5 1.4))
```

We see that two random variables have been generated for parameter x: 1.4037962 and 1.4152595. They are combined in a cross product with the four values of u. The values for y and w are added independently, but the value 0.2, which would be possible for w is never used, because it was discarded when combining the lists of y and w with the minimum independence relation.

Finally, we should mention how the generated values can be used in the program that controls the robot during experience acquisition. The construct `problem-parameters` used in the examples generates a nested list of values as shown. They can be bound to variables by LISP constructs such as `destructuring-bind` and be used in the program. One has to take care of the order in which the values are arranged in the generated list. A more elegant way is provided by the RoLL construct `with-problem-parameters`, whose syntax is similar to the `problem-parameters` construct, but includes a loop and a `let` expression, so that in each run of the loop the parameter variables are bound to new values, which can be used in the body of the construct.

C.4 Abstract Experiences

C.4.1 Experience Definition

`(define-abstract-experience` ⟨experience name $_{\text{symbol}}$⟩
`:specification` ⟨*abstract automaton*⟩
`:experience-class` ⟨*experience class*⟩
`:experience-class-initargs` ⟨*keyword pair*⟩$^{+}$`)`
⟨*experience class*⟩ → `'transient-abstract-experience` | ⟨class name $_{\text{symbol}}$⟩
⟨*keyword pair*⟩ → ⟨key $_{\text{keyword symbol}}$⟩ ⟨value $_{\text{T}}$⟩
⟨*abstract automaton*⟩ → `(`⟨automaton name $_{\text{symbol}}$⟩
`:begin (`⟨variable $_{\text{symbol}}$⟩$^{+}$`)`
`:end (`⟨variable $_{\text{symbol}}$⟩$^{+}$`)`
`:interval (`⟨variable $_{\text{symbol}}$⟩$^{+}$`)`
`:children (`⟨*abstract automaton*⟩$^{+}$`))`

Purpose: Define an abstract experience.

Description: The definition is analogous to the raw experience. This definition only gives the structure of the automaton including the variable names without the data source to the variables. Because an abstract experience can be generated from several raw experience classes (see Figure 5.6 on page 71), the conversion is defined

separately. A shortcut definition for an abstract experience with the conversion is presented below in Section C.4.3.

C.4.2 Experience Conversion

```
(define-experience-conversion
  :from-experience ⟨experience name symbol⟩
  :to-experience ⟨experience name symbol⟩
  :operations ⟨conversion⟩)
```

Purpose: Specify an abstraction step.

Description: Establishes an abstraction between two experiences, the operations being defined in the form of the abstract experience automaton.

Basic Conversion Operations

⟨*conversion automaton*⟩ → ⟨*abstract data automaton*⟩
| `(with-binding` ⟨*binding definition*⟩ ⟨*conversion automaton*⟩`)`
| `(with-filter` ⟨*filter definition*⟩ ⟨*conversion automaton*⟩`)`
| `(with-cross-product` ⟨*cross product definition*⟩ ⟨*conversion automaton*⟩`)`

⟨*abstract data automaton*⟩ → `(`⟨*automaton identifier*[146]⟩
`:begin` ⟨*abstraction data*⟩
`:end` ⟨*abstraction data*⟩
`:interval` ⟨*abstraction data*⟩
`:children` ⟨*abstraction data*⟩`)`

⟨*abstraction data*⟩ → `(`⟨*variable assignment*⟩$^+$`)`

⟨*variable assignment*⟩ → `(`⟨variable $_{\text{symbol}}$⟩ ⟨*value source*⟩`)`

⟨*value source*⟩ → ⟨symbol⟩
| ⟨*automaton var data*⟩
| `(`⟨operator $_{\text{symbol}}$⟩ ⟨*value source*⟩$^+$`)`

Purpose: Operations involved in experience conversion.

Description: The automaton definition is analogous to raw experiences. The source of the variable values is now the content of the raw experience (the experience given in the `from-experience` declaration).

Sophisticated Conversion Operations

⟨*binding definition*⟩ → `(`⟨*variable assignment*⟩$^+$`)`

⟨*filter definition*⟩ → `(`⟨*binding definition*⟩ `:where` ⟨condition $_{\text{LISP expression}}$⟩`)`

⟨*cross product definition*⟩ → `(`⟨*binding definition*⟩ `:where` ⟨condition $_{\text{LISP expression}}$⟩`)`

Purpose: Special operations for abstraction.

Description: These operations make the definition of abstract experiences more comfortable and powerful. In detail, the options are

- local bindings: defining local variables using the values from the raw experience, comparable to a `let` expression;
- filter: a way to omit unwanted experiences, similar to the event handling strategy of the raw experience, but can decide on the basis of abstracted values;
- cross product: specify a set of variables to create the cross product from, thus making several experiences out of one. The result can be restricted to experiences fulfilling the given condition.

C.4.3 Combined Definition of Abstract Experience and Conversion

```
(define-abstract-experience ⟨experience name symbol⟩
  :parent-experience ⟨automaton name symbol⟩
  :specification ⟨conversion automaton[153]⟩
  :experience-class ⟨experience class⟩
  :experience-class-initargs ⟨keyword pair⟩+)
```

Purpose: Compact definition of abstract experience and conversion.

Description: Abbreviation for one-to-one relationships of experiences. The syntax is almost identical to the pure abstract experience definition, the only addition being the parent experience. The automaton is defined like the one in the conversion specification.

C.5 Learning Problems

```
(define-learning-problem
  :function ⟨function specification⟩[1]
  :use-experience ⟨experience name symbol⟩ | ⟨conversion automaton[153]⟩
  :learning-system (⟨class symbol⟩ ⟨keyword pair[152]⟩+)
  :input-conversion ⟨abstraction data[153]⟩
                    | (:generate ⟨conversion⟩+)
  :output-conversion (⟨LISP expression⟩+))
```

Purpose: Define a learning problem.

Description: Specify all parts of a learning problem:

[1] see specification of learning problem class on page 157

- ❏ the function to be learned, the syntax being defined by the learning problem class;
- ❏ the experience, which can either be defined outside the learning problem definition with `define-abstract-experience` or as an anonymous experience inside the learning problem definition;
- ❏ the learning system to be used and the bias given as initialization arguments of the learning system class;
- ❏ the conversion between the function produced by the learning algorithm and the desired one (see below for more explanation).

C.5.1 Function Conversion

The problem of integrating the learned function into the program and the need to rerun the abstraction process is explained on page 97 and in Figure 5.16. Because the inverse abstraction is not known (the conversion from $\langle j,k\rangle$ to $\langle v,w\rangle$ in Figure 5.16), the output conversion is given as a list of LISP expressions, the outer brackets replacing an explicit `progn`. This conversion can return any LISP expression including several return values.

The initial abstraction (from $\langle x,y,z\rangle$ to $\langle h,i\rangle$) can either be given as a list of variable definitions in the style of a `let` expression or it can be generated from the already existing experiences and conversions.

⟨*conversion*⟩ → `(:in-experience` ⟨experience $_{\text{symbol}}$⟩ ⟨*substitution*⟩$^+$`)`
| `(:in-conversion` ⟨raw experience $_{\text{symbol}}$⟩
⟨abstract experience $_{\text{symbol}}$⟩
⟨*substitution*⟩$^+$`)`
⟨*substitution*⟩ → `:set-var` ⟨variable $_{\text{symbol}}$⟩ `:to` ⟨LISP expression⟩
| `:replace` ⟨variable $_{\text{symbol}}$⟩ `:by` ⟨LISP expression⟩

Purpose: Describe the input conversion based on the abstraction chain of the learning experience.

Description: Variable substitutions can take place in any experience or conversion definition. (An abstract experience definition that integrates a conversion specification counts as an experience.) They can either replace the initial definition of a variable or substitute all occurrences of a variable in the abstraction definition (see example below).

Consider the example in Figure C.1. It shows the abstraction chain defined for learning, the learning experience being e_4. There are two experiences as possible sources to e_3. The data of the experiences is shown in a slightly simplified version at the right hand side. When defining a learning problem using e_4 as experience, a possible input conversion is this:

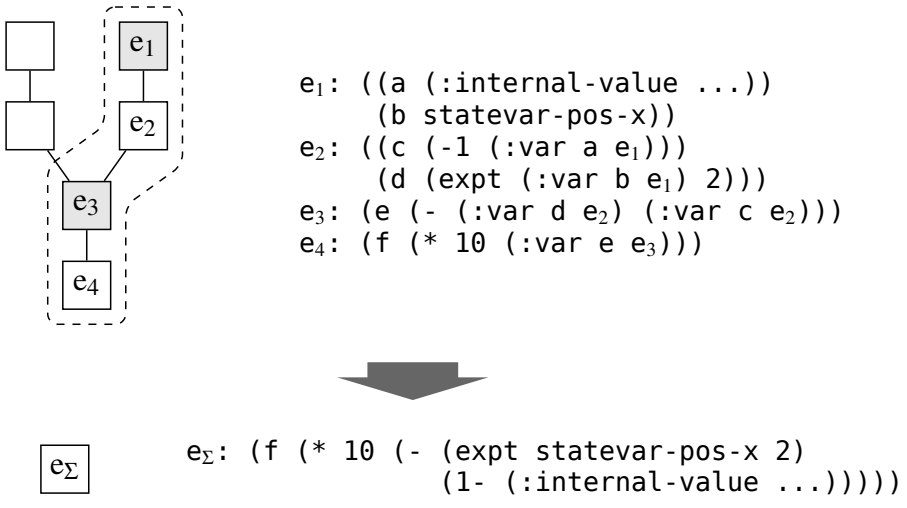
```
(:generate
  (:in-experience e1 :set-var a :to input-value)
  (:in-experience e3 :replace (:var d e2) :by 5))
```

The first thing this definition tells us is that the right branch of the experience tree is interesting for the abstraction generation, because variables in e_1 and e_3 are substituted. The definition must provide a possible solution. It would be incorrect to declare substitutions in both arms of the tree.

With this structural information, a compact experience definition ranging from e_1 to e_4 is generated as shown in the figure. It now remains to substitute the variables. The `:set-var ... :to` construct replaces the right side of a variable definition. This means instead of binding `a` to `(:internal-value ...)` it should be bound to `input-value` instead. The command `:replace ... :by` does a substitution on all occurrences of one expression by another. This means that in the definition of e_3, where `d` is used in a calculation, it is replaced by `5`. The resulting abstraction is

```
(f (* 10 (- 5 (1- input-value))))
```

Figure C.1 Collapsing the course of abstractions into one. If there are several paths of abstraction, the one where the substitutions are defined is chosen.



C.6 RoLL Extensions

C.6.1 Experience Classes

`(define-abstract-experience-class` ⟨class name $_{\text{symbol}}$⟩ `(`⟨superclass $_{\text{symbol}}$⟩$^{+}$`)`
`(`⟨slot definition⟩*`))`

Purpose: Define a new experience class.

Description: The definition is similar to a class definition in LISP. In addition, at least the following methods should be provided for the specified class:

- `deliver-experience`
- `make-conversion-code`

C.6.2 Learning Problem Classes

`(define-learning-problem-class` ⟨lp-class name $_{\text{symbol}}$⟩ `(`⟨superclass $_{\text{symbol}}$⟩$^{+}$`)`
`(`⟨slot definition⟩*`)`
`:definition-schema (`⟨keyword $_{\text{symbol}}$⟩ ⟨argument $_{\text{symbol}}$⟩*`)`
`:name-generation` ⟨LISP expression⟩
`:initargs (`⟨*keyword pair*[152]⟩`)`$^{+}$`)`

Purpose: Define a new learning problem class.

Description: The definition is similar to a class definition in LISP. The definition schema is the pattern after which the function in the learning problem must be defined. The keyword gives an identifier, the rest are arguments. The arguments are bound to slots in the slot definitions whose initargs comply with the argument name, if such slots exist. The initargs declaration establishes the connection between arguments and slots explicitly. The name-generation declaration should contain a functional expression returning a string. The generated name is used as an identifier for the learning problem.

C.6.3 Learning Systems

New learning systems aren't defined by a RoLL construct. Rather, a learning system is a class (derived from the RoLL class `learning-system`) defined with the LISP `defclass` construct and an experience class definition for storing the learning data. To integrate the learning system, the following methods must be added:

- `do-learning`
- `integrate-learned-function`

Appendix D

RoLL Extensions

In Chapter 5 and Appendix C we have introduced the language constructs of RoLL for specifying experiences, their abstraction, and the learning process. We have mentioned several ways to customize and extend the basic functionality of RoLL, so that it can be adapted to new developments in learning algorithms and new kinds of learning problems. These extensions are

- ❏ experience classes,
- ❏ learning problem classes, and
- ❏ learning systems.

In the following, we explain the language constructs of RoLL and the steps to be taken for defining these modules by first presenting the approach and then giving a detailed example. The following sections are also intended as a reference for the set of extensions implemented so far.

D.1 Experience Classes

Experience classes are defined in a class hierarchy as shown in Figure D.1. The general classes of experiences can be classified along a logical line into raw and abstract experiences and along a data processing criterion into transient and persistent experiences. As raw experiences come in as a stream of data, they are defined to be transient. This means, they don't store the data only until they are passed on to the next level of abstraction. We don't expect that new classes of raw experiences are defined by the user, because the process of experience acquisition is very complex and an integral part of RoLL.

Abstract experiences are usually of a persistent character. They can either store experiences for longer times or be the interface for the learning data of the chosen learning algorithm. When there are complex relationships between abstractions, as shown in Figure 5.6 on page 71 and Figure 5.7 on page 72, the use of transient abstract experiences can

be useful. In any case, RoLL allows the definition of new classes of abstract experiences. For permanently storing and managing experiences, we suggest to use the database experience presented below. If no database is available or especially huge sets of experiences are to be stored, log files are an alternative. Examples for abstract experiences containing the data for the learning algorithm are presented in Section D.3.

D.1.1 Process of Defining Experience Classes

For defining a new class of abstract experiences, one has to follow three steps as shown in Figure 5.13 on page 92:

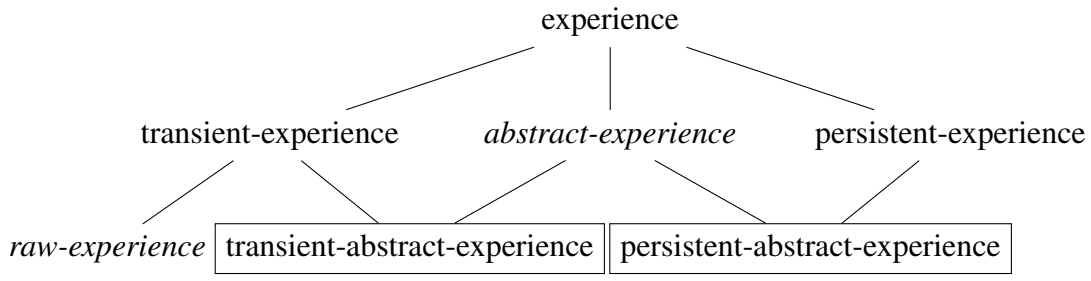
1. define an experience class,
2. specify how incoming data is to be stored,
3. specify how to retrieve the data and transform it to the next abstraction step.

The first step simply consists in defining a LISP class derived either from one of the classes `roll:transient-abstract-experience` for transient experiences or `roll:persistent-abstract-experience` for persistent ones. This class definition may contain any slots necessary for storing or retrieving the data, e.g. a directory where a log file is to be stored.

Secondly, each experience class must define the method `roll:deliver-experience`, which receives the reference instance of the class as input and takes care that the data contained in it is not lost. After calling this method, the reference instance will be used to store other data.

The third step is irrelevant for experiences used as input for the learning system. All other experience classes must provide a method that retrieves an experience and converts it according to the abstraction specification.

Figure D.1 RoLL standard experience classes. The framed classes are the ones new experience classes can be derived from.



D.1.2 Database Experience

For storing experiences permanently, relational databases offer a lot of functionality that makes the management and filtering of experiences possible. For one thing, all experiences are available without having to be read from a file and can be modified either by SQL queries or by data mining mechanisms working directly on the database. Besides, additional information concerning the experience instances can be stored, e.g. the time when the experience was made, so that newer experiences can be trusted more than older ones. Moreover, there are lots of tools and frontends available to visualize the contents of SQL databases, which helps to get a better understanding of experiences.

The definition of a database experience presented in the following is not the only way of defining an abstract experience using a relational database as storage method. It uses a specific way of linking LISP to the database and defines a specific mapping from experiences to tables. Besides, it stores the time when an experience was made as additional management information.

Class Definition

```
(roll:define-abstract-experience-class
      database-experience (roll:persistent-abstract-experience)
 ((table-name-prefix :accessor table-name-prefix)
  (database-spec     :accessor database-spec)
  (db-signature-set :initform nil :accessor db-signature-set)
  (db-update-fun    :initform nil :accessor db-update-fun)
  (last-episode-nr  :initform nil :accessor last-episode-nr)))
```

The class `database-experience` contains information about the database access, a naming convention for easily finding the tables produced by LISP in the SQL tables, and information for converting LISP to SQL types and guaranteeing unique keys.

Storing Data

```
(defmethod roll:deliver-experience ((experience database-experience))
 (let ((data (funcall (eval (db-update-fun experience)) experience)))
  (clsql:with-database (db (database-spec experience))
   (mapcar #'(lambda (str) (clsql:execute-command str :database db))
           data))))
```

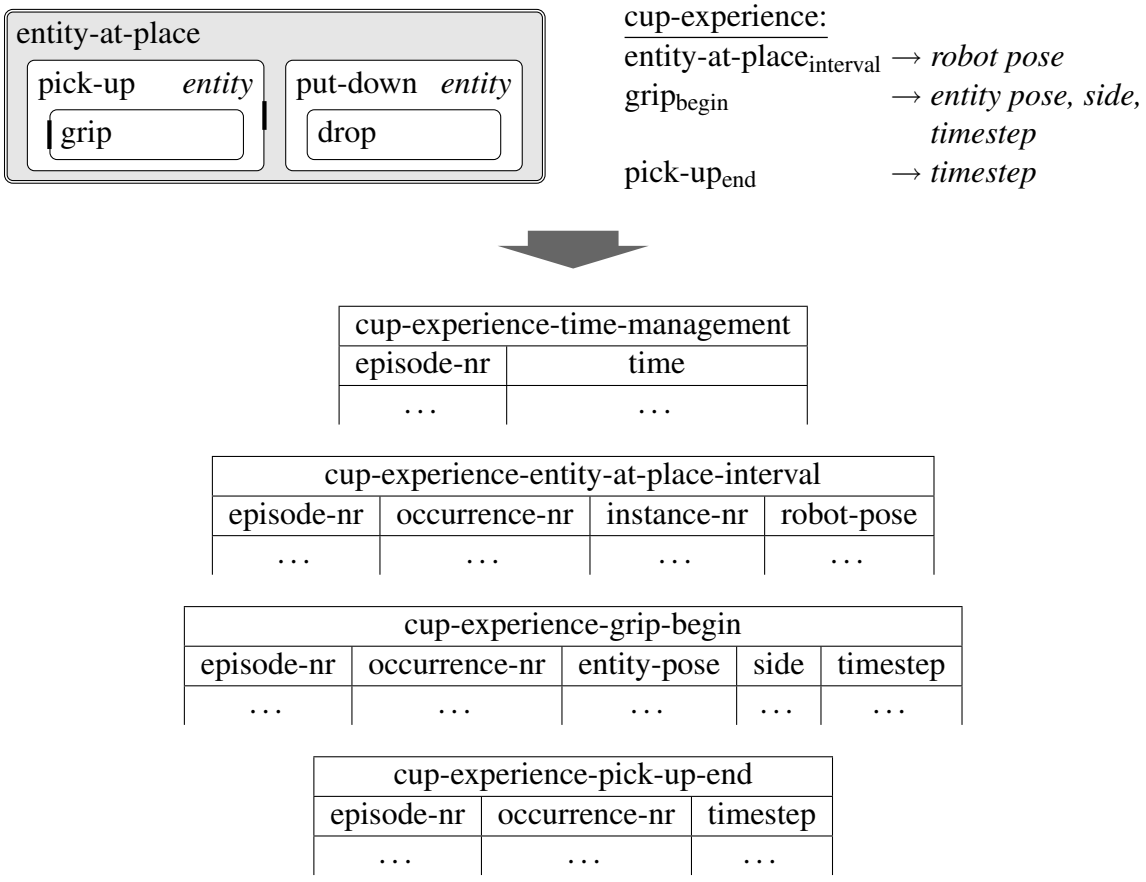
For storing experiences in a database, the first design decision is how to map the automaton structure of the LISP experience class with *begin*, *end* and *interval* slots to database tables. Figure D.2 shows how tables are created from an experience definition. For each data chunk (the data recorded in one automaton at one of the times begin, end, or interval) one table is created, whose name is composed of

- ❏ the name of the experience (in the example *cup-experience*),
- ❏ the name of the automaton (entity-at-place, grip, pick-up),
- ❏ the recording time (begin, end, interval).

The signature of the table always contains the episode number (a running number counting the number of instances available of this experience type) and the occurrence number (according to the definition in addressing experience data explained on page 70). For interval tables, the instance number of the data set is also stored. The rest of the table structure is computed from the variables that are to be recorded according to the names provided in the experience definition. The SQL data types are derived automatically from the LISP types when the first experience is to be stored, although this may be ambiguous (especially the LISP value `nil` can be translated to `false` or `null`, so that the data type can either be Boolean — this is what we chose — or any SQL type).

Beside the tables generated from the experience definition, the database experience

Figure D.2 Mapping between automaton structure and database tables.



adds another table, whose name is composed of the experience name and the suffix *-time-management*, where the episode number and the time when the experience was made are stored.

Retrieving and Converting Data

```
(defmethod roll:make-conversion-code ((from database-experience)
                                      (to experience)
                                      operations from-var to-var
                                      &optional)
  ...)
```

The simplest way to convert database experiences into other classes of experiences is to read an episode from the database, store it in the reference instance of the experience class in LISP and then use the standard conversion from one experience instance to another one. However, databases offer a wide variety of operations that can help to reduce the data transferred between the LISP and database processes and make computations faster. One basic operation of databases is calculating the cross product of tables. Therefore, if the abstraction contains a cross product operation, it is executed directly in the database. Although other operations, like arithmetic, are possible as well, we decided to perform those in LISP.

The method `make-conversion-code` should return code that generates the method `convert` like a macro, because it depends on the automaton and data structure of the experience.

D.2 Learning Problem Classes

For RoLL it is important to know what kind of function is to be learned, e.g. a prediction model for a routine or the function choosing commands for a low-level routine. Different kinds of functions need different parameters to be defined unambiguously. For example, for learning a low-level routine, the name of the routine must be given. When a model of a routine is to be learned, not only the routine, but also the type of the model (e.g. time prediction) must be specified.

This information is necessary for integrating the learning result appropriately into the program. The integration is done in accordance with the learning system, which means that the generic function `integrate-learned-function` is determined by the learning system and the learning problem class. Here we only show the definition of learning problem classes, an example of integrating learned functions is described in the next section.

The two examples in Listing D.1 show how a learning problem class is to be defined. The specification looks similar to a LISP class definition. After the name the parent classes are given, usually this is the RoLL class `learning-problem`. Then slots containing the parameters of the learning problem class can be defined. The definition schema specifies the way in which the learning problem class is chosen when specifying a learning problem (compare the definition of a learning problem in Section 6.3 on page 109). The variables in this schema can either correspond to the initargs of the slots defined in the learning problem class or a mapping between the variables in the definition scheme and the initargs of the slots can be established with the parameter `:initargs`.

The parameter `:name-generation` describes how to generate a unique name for a learning problem instance of the specified class. For example, the learning problem shown in Listing 6.3 on page 109 can be addressed by the name `b21-go2pose-time-model`.

D.3 Learning Systems

The most important dimension of extending RoLL is the possibility to use any experience-based learning algorithm. For RoLL it is of no importance if the learning system is implemented in LISP or in an external program as long as it can be called from LISP. The main differences between learning systems are

- the bias (i.e. the parameters controlling the learning process),
- the format of the input data, and
- the output format.

The procedure of defining a new learning system is oriented along these differences:

Listing D.1 Definition of learning problem classes.

```
; prediction models for routines
(roll:define-learning-problem-class model-learning-problem
                                    (roll:learning-problem)
 ((routine    :initarg :routine :initform nil :accessor routine)
  (model-type :initarg :type    :initform nil :accessor model-type))
 :definition-schema (:model routine type)
 :name-generation   (format nil "~a-~a-model" routine model-type))

; low-level routines
(roll:define-learning-problem-class routine-learning-problem
                                    (roll:learning-problem)
 ((rname :initarg :rname :initform nil :accessor rname))
 :definition-schema (:routine routine-name)
 :name-generation   (format nil "~a-routine" routine-name)
 :initargs          (:rname routine-name))
```

1. Define a class derived from `roll:learning-system`, containing all the parameterization needed for running the learning process. This class is the interface for choosing and parameterizing the learning system later.
2. Define an experience class whose storage format fits the requirements of the learning system. For a learning experience class, only steps 1 and 2 of the procedure described on page 159 are necessary since the data needn't be read back into the RoLL system.
3. Implement the invocation of the learning process with the bias given by the user.
4. Specify how to integrate the result of the learning process into LISP. This step consists of two problems: conversion of the output from the learning system to executable LISP code, and integration of this code into the program in a way that fits the chosen learning problem class. As these two steps are strongly interwoven, they have to be specified in one method `integrate-learned-function`. This requires knowledge of both the learning system and the learning problem class.

Although RoLL doesn't assume a specific format for the output of the learning system, it requires that the result of the learning process be written to a LISP file, so that it can be loaded in later runs of the system. Therefore, each learning problem is provided the information of where to put the file containing the learned function (a slot inherited from the class `roll:learning-system`, the information being provided by the programmer) and the LISP package of the function to be learned. Besides, a unique identifier for each learning problem is generated according to the rules of the learning problem.

In the following we present the learning systems implemented so far. Both make use of external programs. The first employs several decision tree algorithms of the WEKA machine learning software, the second provides access to the Stuttgart Neural Network Simulator (SNNS).

D.3.1 Learning System WEKA

The WEKA (Witten and Frank 2005) machine learning project is a toolkit providing different learning and data mining algorithms implemented in Java. We only include the algorithms for decision tree learning, these are J48 for classical decision trees and M5' for model and regression trees (Belker 2004).

Learning System Class

As the WEKA software supports a variety of learning algorithms, we arrange the learning algorithms in a class hierarchy similar to the structure in WEKA. This approach simplifies the integration of new WEKA algorithms and exploits features of the class hierarchy. Figure D.3 depicts the hierarchy graphically, whereas Listing D.2 shows the code for

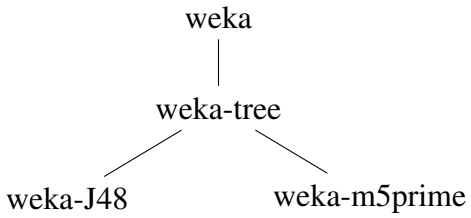
defining these classes in RoLL.

The input for all WEKA classifiers is given in the attribute-relation file format (arff) and the result is stored in an output file with the extension `.weka`, whose format differs however when using different learning algorithms. The output produced by WEKA is no executable code, but must be converted to a function in some programming language. Once this conversion is done, the WEKA output file can be deleted or be kept for inspection. This decision can be controlled by a flag in the `weka` class.

The next step in defining the classes for decision tree learning is a class called `weka-tree`. The class definition subsumes parameters that are valid in all tree learning algorithms. Besides, the output format of the different tree learning algorithms is similar, though not identical. By defining the `weka-tree` class, some of the parsing can be handled on the level of the general tree class.

Finally, we define two classes `weka-J48` and `weka-m5prime` for different decision tree learning algorithms. The differences are mainly in the call to WEKA and the parsing of the resulting tree given in the WEKA result file. The J48 algorithm is an enhancement of the well-known C4.5 algorithm and trains classical decision trees. The M5' algorithm handles both model trees and regression trees, the choice being made by the flag `build-regression-tree`.

Figure D.3 Class hierarchy for WEKA tree learning algorithms.



WEKA Experience

The input format to all WEKA classifiers is independent of the chosen algorithm. All classifiers assume that the data is composed of input and output values. Therefore, the automaton structure of a weka-experience assumes that only one automaton with *begin* and *end* values and no *interval* values is given.

For defining input data files WEKA requires a name for the relation to be learned, the types of the attributes (input and output), and the data itself. The relation name is stored in the `weka-experience` class (see Listing D.3) and is set automatically to the generated name of the learning problem. The types of the attributes must be specified manually. They could be set automatically like the SQL data types of the database expe-

Listing D.2 Class definitions for WEKA tree learning algorithms.

```
;; all learning algorithms supported by WEKA
(defclass weka (roll:learning-system)
 ((arff-file        :initarg :arff-file :reader arff-file)
  (weka-output-file :initarg :weka-output-file
                    :reader weka-output-file)
  (delete-weka-output-file :initarg :delete-weka-output-file
                           :initform nil
                           :reader delete-weka-output-file)))

;; tree learning algorithms
(defclass weka-tree (weka)
 ((use-unpruned-tree :initarg :use-unpruned-tree
                     :initform nil
                     :accessor use-unpruned-tree)
  (minimum-number-of-instances
   :initarg :minimum-number-of-instances
   :initform nil
   :accessor minimum-number-of-instances)))

;; classical decision tress
(defclass weka-J48 (weka-tree)
 ((pruning-confidence-threshold
   :initarg pruning-confidence-threshold
   :initform nil
   :accessor pruning-confidence-threshold)
  (reduced-error-pruning :initarg reduced-error-pruning
                         :initform nil
                         :accessor reduced-error-pruning)
  (number-of-folds :initarg number-of-folds
                   :initform nil
                   :accessor number-of-folds)
  (use-binary-splits-only :initarg use-binary-splits-only
                          :initform nil
                          :accessor use-binary-splits-only) ))

;; regression and model trees
(defclass weka-m5prime (weka-tree)
 ((use-unsmoothed-predictions :initarg :use-unsmoothed-predictions
                              :initform nil
                              :accessor use-unsmoothed-predictions)
  (build-regression-tree :initarg :build-regression-tree
                         :initform nil
                         :accessor build-regression-tree)))
```

rience presented in Section D.1.2, but the mapping is sometimes ambiguous and can only be determined after the first experience instance (i.e. the actual data) is known.

The conversion of the learning data to the attribute-relation file format works identically to the conversion from an experience to a more abstract one. The experience data is converted episode-wise as shown in Figure 5.13 on page 92. When the reference instance of the weka-experience is filled with data, its content is written to a file in the required format. The writing of the attribute-relation file is done in two steps: first only the data is written to a file `arff-tmp-file` specified in the WEKA experience class. The final data file is created before the learning process starts (see next section).

Performing the Learning Process

Invoking the WEKA learning software from RoLL is very simple. The two steps to be performed for all algorithms provided by WEKA is to complete the missing information (the attribute names and types) in the data input file and then to call WEKA with the chosen algorithm. This call to WEKA differs for each algorithm. It must contain the correct WEKA class and add the specified parameters to the method invocation. The output of the learning process is written to the output file specified in the learning problem specification. The definition of the general learning procedure for all WEKA algorithms and the specific call to the M5' algorithm are shown in Listing D.4.

Listing D.3 Definition of experience class for WEKA algorithms. The data is stored in the attribute-relation file format required by WEKA.

```
;; weka-experience class
(roll:define-abstract-experience-class weka-experience
                                       (roll:learning-experience)
 ((arff-tmp-file   :initform nil :accessor arff-tmp-file)
  (relation-name   :initform nil :accessor relation-name
                   :initarg :relation-name)
  (attribute-types :initform nil :accessor attribute-types
                   :initarg :attribute-types)))

;; storing experience data
(defmethod roll:deliver-experience ((experience weka-experience))
 (ensure-directories-exist (arff-tmp-file experience))
 (with-open-file (stream (arff-tmp-file experience)
                  :direction :output :if-exists :append
                  :if-does-not-exist :create)
  (format stream "~{~,5f,~}~{~,5f~^,~}~%"
   (roll:get-automaton-data experience :begin)
   (roll:get-automaton-data experience :end))))
```

Integrating the Learning Results

The most laborious part in the definition of a new learning system is the integration of the learning result back into RoLL. In the case of WEKA there are no tools translating the output of the learning process to a C or LISP function, which could easily be embedded into the program. Instead, the tree learning algorithms produce a well-readable text file containing the tree structure. We parse this tree in LISP and return executable code, which must then be written to a file either in the form of a LISP function or method or a lambda function in the correct context, according to the learning problem class.

Listing D.4 Definition of learning process for WEKA algorithms, in particular M5'.

```
;; invocation of WEKA classification algorithm
(defmethod roll:do-learning ((ls weka) (experience weka-experience))
 ;finish arff file
 (port:run-prog "/bin/bash"
  :args (list
         "-c"
         (format
          nil
          "echo -e \"@relation ~a\\n\\n~{@attribute ~{~a~^ ~}\\n~}
           \\n@data\"; cat ~a"
          (relation-name experience)
          (attribute-types experience)
          (namestring (arff-tmp-file experience))))
  :output (arff-file ls)
  :if-output-exists :supersede)
 (port:run-prog "rm"
   :args (list (namestring (arff-tmp-file experience))))
 ;run specific learning algorithm
 (run-weka ls))

;; call to WEKA for M5' algorithm
(defmethod run-weka ((ls weka-m5prime))
 (port:run-prog "java"
  :args (append
         '("weka.classifiers.trees.M5P")
         (when (use-unpruned-tree ls) '("-N"))
         (when (use-unsmoothed-predictions ls) '("-U"))
         (when (build-regression-tree ls) '("-R"))
         (when (minimum-number-of-instances ls)
           (list (format nil "-M ~a"
                         (minimum-number-of-instances ls))))
         `("-t" ,(namestring (arff-file ls))))
  :output (weka-output-file ls) :if-output-exists :supersede))
```

The solution for integrating WEKA results into the RoLL program is shown in Listing D.5. The method `integrate-learned-function` is specialized to the WEKA learning system and any kind of learning problem. It parses the WEKA output and produces a general frame for the LISP file that is to contain the resulting function.

The function definition according to the learning problem class is done by the method `make-weka-function-call`, which takes the learning problem object and the function body generated by parsing the WEKA output file. In Listing D.5 we show the integration for the learning problem class for routine models presented in Section D.2. In this case the function code is surrounded by a lambda function, which is placed in the model slot of the routine specified in the learning problem.

The resulting file is loaded directly after the learning process and then every time the LISP system is started. A prerequisite is of course that the learned file is written to a directory where it is loaded automatically. This can easily be achieved by loading files with the ASDF system.

Listing D.5 Integration of WEKA results when learning a prediction model for a routine.

```
(defmethod roll:integrate-learned-function
                ((ls weka) (lp roll:learning-problem))
 (with-open-file (str (learned-function-file ls)
                      :direction :output :if-exists :supersede)
  (format str "(in-package ~s)~2%"
          (package-name (learned-function-package ls)))
  (format str "~s"
          (make-weka-function-call
           lp
           `(let ((,(first (second (learning-signature ls)))
                   ,(parse-weka-output ls)))
              ,@(output-conversion lp)))))
 (when (delete-weka-output-file ls)
  (port:run-prog "rm"
   :args (list (namestring (weka-output-file ls))))))

; make-weka-function-call for model-learning problem
(defmethod make-weka-function-call ((lp model-learning-problem)
                                    function-body)
 (let ((routine-var (gentemp "ROUTINE")))
  `(add-model ',(routine lp) ,(model-type lp)
    (make-instance 'routine-model
     :execution-entity #'(lambda (,routine-var)
                           (let* ,(make-learning-conversion-code
                                   (input-conversion lp)
                                   `((:input ,routine-var)))
                            ,function-body))))))
```

D.3.2 Learning System SNNS

SNNS (Stuttgart Neural Network Simulator) is an interface for defining and training neural nets. It supports a great variety of network architectures and learning algorithms. The interface presented here only considers layered feed-forward networks. Next to a graphical user interface, SNNS offers a batch mode and tools for creating network structures on the command line. We use the latter method for making SNNS usable in RoLL.

The steps for defining SNNS as a RoLL learning system are the same as for the WEKA tree learning algorithms just presented. We will therefore omit the code and only explain the most important aspects.

Learning System Class

The bias of neural networks contains numerous parameters. Even when restricting the interface to layered feed-forward networks as we do, SNNS offers lots of possibilities for customization, which are described in detail by Zell and others (1998). The following parameters are supported in the RoLL interface (we refer to the pages of the SNNS manual in brackets):

- unit activation fun (pp. 316, 249)
- unit output fun (p. 317)
- net initialization function (pp. 82, 245)
- net learning function (pp. 67 ff., 145ff., 246)
- number of cycles to train the network
- hidden layers

Besides, for running SNNS a number of files are required:

- a pattern file for storing the learning data,
- a network file containing the structure of the network,
- a batchman file containing instructions for SNNS what to do,
- a result file, which is a copy of the pattern file with additional information about the errors for each training example,
- a C file containing the learned network as a C function,
- a library file — the compiled C file.

The use of these files is explained below. The most important consideration when providing these paths is that the library file, i.e. the compiled result file, must be placed in a directory where it is loaded by the RoLL system at start-up, so that the learned function is available every time the system is started.

SNNS Experience

SNNS expects the input data in form of a so-called pattern file. Next to the learning data it contains information about the number of learning examples and the signature, i.e. which of the given values are to be used for input and which ones represent output variables. The input and output variables are drawn from the experience definition, which must contain one automaton with the input variables in the begin slot and the output variables in the end slot.

In contrast to the WEKA experience, the SNNS experience format doesn't require additional information like the types of the input values, which are assumed to be floating point numbers. The method of writing the patterns is very similar to the WEKA experience format. The additional information about the number of patterns is likewise added just before the learning process starts.

Performing the Learning Process

For running SNNS, three files must be created out of the parameters given in the learning system class: a pattern file, a network file, and a batchman file.

The pattern file is finished when the learning starts, but is mostly composed of the data written in the experience conversion process.

The network file describes the structure of the network. The number of input and output variables is computed from the experience, whereas the number of hidden layers and the number of neurons contained in them is given in the learning system parameters. Besides, the user-defined unit activation and output functions are set in the network. Other network structures are supported by SNNS, but not by this RoLL interface.

The batchman file tells SNNS how to perform the learning process. SNNS offers a rich language for controlling the learning process and evaluating the result. However, we implemented a simple loop training the network as many times as given in the learning system specification and then stop. Beside the trained network we also save the result file, so that the error values can be checked manually. If a learning problem requires a more sophisticated control, the interface can still be used without creating the batchman file automatically, but using a hand-coded batch program.

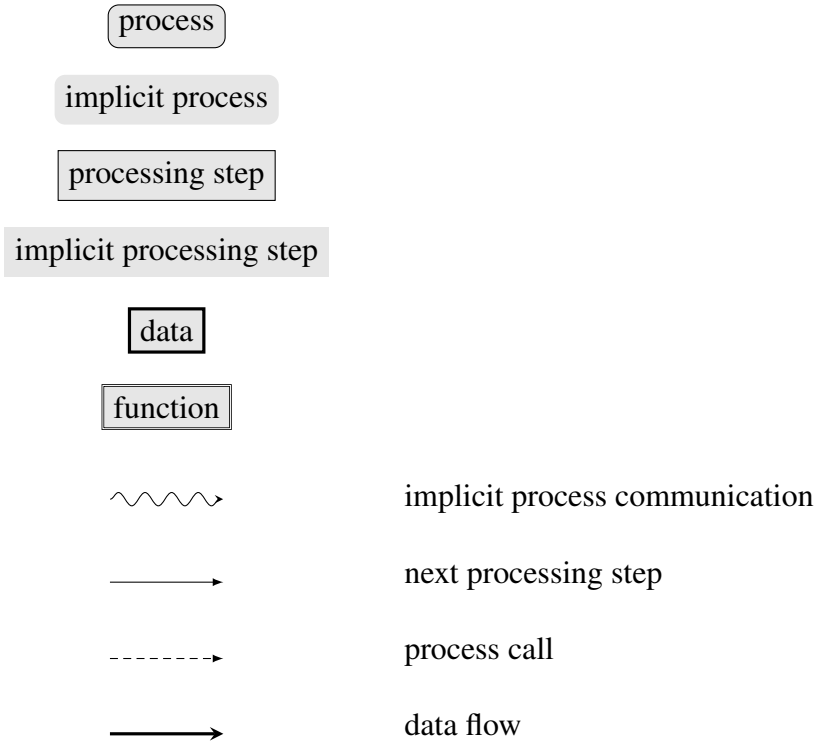
Integrating the Learning Results

When the learning process is done, the trained network is saved in a format specific to SNNS. This file can be converted to C with the tool SNNS2C. The generated C function is then compiled to a shared object library, which can be integrated into LISP using the universal foreign function interface (UFFI). The foreign function call is then added to the RoLL program according to the chosen learning problem class.

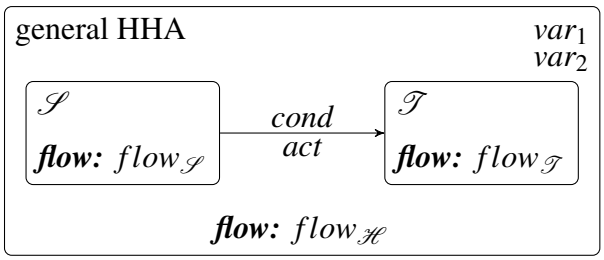
List of Figures

Drawing Conventions

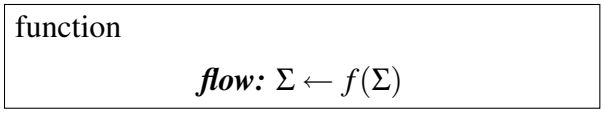
Control Flow and Architecture Diagrams



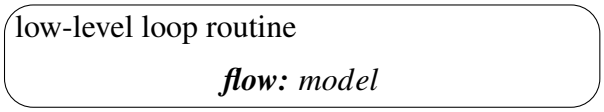
Hybrid Automata



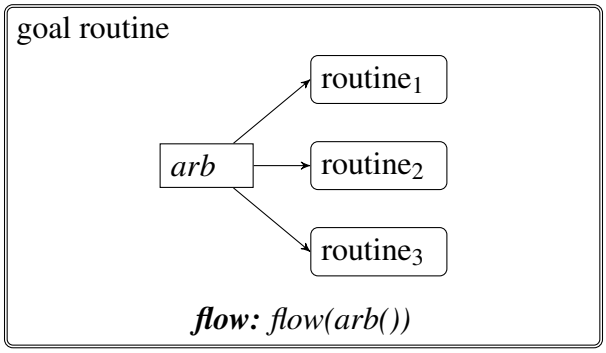
Hybrid automaton with subautomata $\mathscr{S}$ and $\mathscr{T}$, variables *var*1 and *var*2, a transition $\langle \mathscr{S}, \mathscr{T} \rangle$ with jump condition and *act* function, and a $flow$ function.



Function of the BDI language layer, shown as an automaton with sharp corners.



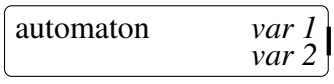
Low-level loop routine of the BDI layer, illustrated by rounded corners with a larger radius. Other low-level routines are depicted as standard automata.



Goal of the BDI language layer, depicted as an automaton with double lines.

Experience automaton recording data continuously during the automaton run.

Experience automaton recording data once at the beginning of the automaton activity.



Experience automaton recording data once at the end of the automaton activity.

Bibliography

Abraham, Ajith (2003). Meta-learning evolutionary artificial neural networks. *Neurocomputing* .

Abraham, Ajith and Baikunth Nath (2000). Optimal design of neural nets using hybrid algorithms. In *Pacific Rim International Conference on Artificial Intelligence*, pp. 510–520.

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.

Alu, R., J. Esposito, M. Kim, V. Kumar, and I. Lee (1999). Formal modeling and analysis of hybrid systems: A case study in multirobot coordination. In *Proceedings of the World Congress on Formal Methods*, pp. 212–232.

Alur, R., C. Belta, F. Ivancic, V. Kumar, M. Mintz, G. Pappas, H. Rubin, and J. Schug (2001). Hybrid modeling and simulation of biomolecular networks. In *Fourth International Workshop on Hybrid Systems: Computation and Control*, pp. 19–32.

Alur, R., C. Courcoubetis, T. A. Henzinger, and P-H. Ho (1993). Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. *Lecture Notes in Computer Science* 736: 209–229.

Alur, R., T. Dang, and F. Ivancic (2002). Reachability analysis of hybrid systems via predicate abstraction. In *Fifth International Workshop on Hybrid Systems: Computation and Control*.

Alur, R., T. Henzinger, and P. Ho (1996). Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering* 22(3): 181–201.

Alur, Rajeev, Radu Grosu, Insup Lee, and Oleg Sokolsky (2001). Compositional refinement for hierarchical hybrid systems. *Lecture Notes in Computer Science* 2034.

Alur, Rajeev, Thomas A. Henzinger, and Pei-Hsin Ho (1996). Automatic symbol verification of embedded systems. *IEEE Transactions on Software Engineering* 22(3).

Anderson, Michael L., Tim Oates, Waiyian Chong, and Donald R. Perlis (2006). Enhancing reinforcement learning with metacognitive monitoring and control for improved perturbation tolerance. *Journal of Experimental and Theoretical Artificial Intelligence* 18(3).

Anderson, Scott D., David L. Westbrook, David M. Hart, and Paul R. Cohen (1994). *Common Lisp Interface Package CLIP*.

Andre, D. and S. Russell (2001). Programmable reinforcement learning agents. In *Proceedings of the 13th Conference on Neural Information Processing Systems*, pp. 1019–1025, Cambridge, MA. MIT Press.

Andre, David (2003). Programmable Reinforcement Learning Agents. Ph.D. diss., University of California at Berkeley.

Antsaklis, Panos J. (2000). A brief introduction to the theory and applications of hybrid systems. *Proceedings of the IEEE* 88(7): 879–887.

Bachmann, Marc (2007). Concurrent execution of robot plans for every-day activities using learned prediciton models. Master's thesis, Technische Universität München.

Barrett, A. (2003). Domain compilation for embedded real-time planning. In *Proceedings of the ICAPS'03 Workshop on Plan Execution*.

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, 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 and Henrik Grosskreutz (2005). Probabilistic hybrid action models for predicting concurrent percept-driven robot behavior. *Journal of Artificial Intelligence Research* 24: 799–849.

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)*.

Belker, T. (2004). Plan Projection, Execution, and Learning for Mobile Robot Control. Ph.D. diss., Department of Applied Computer Science, University of Bonn.

Blumberg, Bruce (2002). *Exploring Artificial Intelligence in the New Millennium*, chapter D-Learning: What Learning in Dogs Tells Us About Building Characters That Learn What They Ought to Learn. Morgan Kaufmann.

Bonarini, Andread, Alessandro Lazaric, Marcello Restelli, and Patrick Vitali (2006). Self-development framework for reinforcement learning agents. In *Proceedings of the Fifth International Conference on Development and Learning*.

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*.

Bonnlander, Brian (1996). Nonparametric selection of input variables for connectionist learning. Ph.D. diss., University of Colorado.

Boutilier, Craig, Raymond Reiter, Mikhail Soutchanski, and Sebastian Thrun (2000). Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence*, pp. 355–362.

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

Branicky, Michael S. (1995). Studies in Hybrid Systems: Modeling, Analysis, and Control. Ph.D. diss., Massachusetts Institute of Technolgy.

Branicky, M.S. (1993). Topology of hybrid systems. In *Proceedings of the 32nd IEEE Conference on Decision and Control*, pp. 2309–2314.

Branicky, M.S., V.S. Borkar, and S.K Mitter (1998). A unified framework for hybrid control: model and optimal control theory. *IEEE Transactions on Automatic Control* 43(1): 31–45.

Bratman, Michael E. (1987). *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA.

Bratman, Michael E., David Israel, and Martha Pollack (1991). Plans and resource-bounded practical reasoning. In Cummins, Robert and John L. Pollock, editors, *Philosophy and AI: Essays at the Interface*, pp. 1–22. The MIT Press, Cambridge, Massachusetts.

Brodley, Carla E. (1995). Recursive automatic bias selection for classifier construction. *Machine Learning* 20(1–2): 63–94.

Buss, Martin, Michael Beetz, and Dirk Wollherr (2007). CoTeSys — cognition for technical systems. In *Proceedings of the 4th COE Workshop on Human Adaptive Mechatronics (HAM)*.

Caruana, Rich and Dayne Freitag (1994). Greedy attribute selection. In *International Conference on Machine Learning*, pp. 28–36.

Cohn, David A. (1996). Neural network exploration using optimal experiment design. *Neural Networks* 9(6): 1071–1083.

Crites, Robert H. and Andrew G. Barto (1996). Improving elevator performance using reinforcement learning. In Touretzky, D.S̃. M.C̃. Mozer, and M.Ẽ. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*. MIT Press.

Crowley, James L., Olivier Brdiczka, and Patrick Reignier (2006). Learning situation models for understanding activity. In *Proceedings of the Fifth International Conference on Development and Learning*.

Dewey, John (1938). *Experience and Education*. Collier Books, New York.

Dousson, Christophe (1994). Suivi d'Evolutions et Reconnaissance de Chroniques. Ph.D. diss., Université Paul Sabatier, Toulouse.

Dousson, Christophe, Paul Gaborit, and Malik Ghallab (1993). Situation recognition: representation and algorithms. In *13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, pp. 166–172.

Fei-Fei, Li (2006). Knowledge transfer in learning to recognize visual objects classes. In *Proceedings of the Fifth International Conference on Development and Learning*.

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, Malik Ghallab, Guillaume Infantes, and Derek Long (2006). Robot introspection through learned hidden markov models. *Artificial Intelligence* 170(2): 59–113.

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

Gabel, Thomas, Roland Hafner, Sascha Lange, Martin Lauer, and Martin Riedmiller (2006). Bridging the gap: Learning in the RoboCup simulation and midsize league. In *Proceedings of the 7th Portuguese Conference on Automatic Control*.

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

Gehrke, Johannes and Samuel Madden (2004). Query processing in sensor networks. *IEEE Pervasive Computing* 3(1): 46–55.

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*.

Georgeff, Mike, Barney Pell, Martha Pollack, Milind Tambe, and Mike Wooldridge (1999). The belief-desire-intention model of agency. In Müller, Jörg, Munindar P. Singh, and Anand S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, Vol. 1555, pp. 1–10. Springer-Verlag: Heidelberg, Germany.

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 (ICAR2003)*, pp. 317–323.

Ghallab, Malik (1998). Chronicles as a practical representation for dealing with time, events and actions. In *6th Italian Conference on Artificial Intelligence (AIIA'98)*, pp. 6–10.

Gordon, Diana F. and Marie Desjardins (1995). Evaluation and selection of biases in machine learning. *Machine Learning* 20(1–2): 5–22.

Henry, Melvin Michael (2002). Model-based estimation of probabilistic hybric automata. Master's thesis, Massachusetts Institute of Technology.

Henzinger, Thomas (1996). The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pp. 278–292, New Brunswick, New Jersey.

Henzinger, Thomas A. and Howard Wong-Toi (1996). Using hytech to synthesize control parameters for a steam boiler. In *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, pp. 265–282.

Herrera, L. J., H. Pomares, I. Rojas, M. Verleysen, and A. Guilén (2006). Effective input variable selection for function approximation. In *International Conference on Artificial Neural Networks*, Lecture Notes in Computer Science, pp. 41–50. Springer.

Infantes, Guillaume, Felix Ingrand, and Malik Ghallab (2006). Learning behaviors models for robot execution control. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI)*.

Ingham, M., R. Ragno, and B. C. Williams (2001). A reactive model-based programming language for robotic space explorers. In *International Symposium on Artificial Intelligence, Robotics, and Automation in Space (i-SAIRAS)*, Montreal, Canada.

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 (2005). Towards high-performance robot plans with grounded action models: Integrating learning mechanisms into robot control languages. In *ICAPS Doctoral Consortium*.

Kirsch, Alexandra and Michael Beetz (2005). Combining learning and programming for high-performance robot controllers. In *Autonome Mobile Systeme 2005*.

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.

Kirsch, Alexandra, Michael Schweitzer, and Michael Beetz (2005). Making robot learning controllable: A case study in robot navigation. In *Proceedings of the ICAPS Workshop on Plan Execution: A Reality Check*.

Knight, Russell, S. Chien, and Gregg Rabideau (2001). Extending the representational power of model-based systems using generalized timelines. In *The 6th International Symposium on Artificial Intelligence, Robotics, and Automation in Space (i-SAIRAS)*, Montreal, Canada.

Kolb, D. (1984). *Experiential Learning: Experience as the Source of Learning and Development*. Financial Times Prentice Hall, New York, NY.

Laborie, Philippe and Malik Ghallab (1995). IxTeT: an integrated approach for plan generation and scheduling. In *4th IEEE-INRIA Symposium on Emerging Technologies and Factory Automation (ETFA'95)*, pp. 485–495.

Lange, Sascha and Martin Riedmiller (2005). Evolution of computer vision subsystems in robot navigation and image classification tasks. In D. Nardi, M. Riedmiller, C. Sammut and J. Santos-Victor, editors, *RoboCup-2004: Robot Soccer World Cup VIII*. Springer, LNCS.

Li, Yifan, Petr Musilek, and L. Wyard-Scott (2004). Fuzzy logic in agent-based game design. In *Proceedings of the 2004 Annual Meeting of the North American Fuzzy Information Processing Society*, pp. 734–739.

Madden, Sam R., Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong (2005). TinyDB - an acquisitional query processing system for sensor networks. *ACM Trans. Database Systems* 30(1).

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

McDermott, Drew (1990). Planning reactive behavior: A progress report. In *Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 450–458.

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*.

Minsky, Marvin (2001). It's 2001. where is hal? Game Developers Conference Talk, `http://technetcast.ddj.com/tnc_play_stream.html?stream_id=526`.

Mishap Investigation Board (1999). Mars climate orbiter. Phase I Report.

Mitchell, Tom M. (1990). Becoming increasingly reactive. In *National Conference on Artificial Intelligence*.

Mitchell, Tom M. (2006). The discipline of machine learning. Technical report CMU-ML-06-108, Carnegie Mellon University.

Mukerjee, Amitabha and Mausoom Sarkar (2006). Perceptual theory of mind: An intermediary between visual salience and noun/verb acquisition. In *Proceedings of the Fifth International Conference on Development and Learning*.

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, 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 International Conference on Automated Planning and Scheduling (ICAPS)*.

Muscettola, Nicola, P. Pandurang Nayak, B. Pell, and B. Williams (1998). Remote Agent: To boldly go where no AI system has gone before. *Artificial Intelligence* 103(1-2): 5–48.

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

Myers, Karen (2006). Building an intelligent personal assistant. AAAI Invited Talk.

O'Sullivan, Joseph, Karen Zita Haigh, and G. D. Armstrong (1997). *Xavier*.

Oudeyer, Pierre-Yves, Frédéric Kaplan, and Verena V. Hafner (2007). Intrinsic motivation systems for autonomous mental development. *IEEE Transactions on Evolutionary Computation* 11(2): 265–286.

Oudeyer, Pierre-Yves, Frédéric Kaplan, Verena V. Hafner, and Andrew Whyte (2005). The playground experiment: Task-independent development of a curious robot. In *proceedings of the AAAI Spring Symposium Workshop on Developmental Robotics*.

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

Petkos, Georgios, Marc Toussaint, and Sethu Vijayakumar (2006). Learning multiple models of non-linear dynamics for control under varying contexts. In *International Conference on Artificial Neural Networks*, Lecture Notes in Computer Science, pp. 898–907. Springer.

Pfahringer, Bernhard, Hilan Bensusan, and Christophe Giraud-Carrier (2000). Meta-learning by landmarking various learning algorithms. In *Proceedings of the Seventeenth International Conference on Machine Learning, ICML'2000*, pp. 743–750, San Francisco, California. Morgan Kaufmann.

Pfeffer, Avi (2001). IBAL: A probabilistic rational programming language. In *IJCAI*, pp. 733–740.

Pollack, Martha E. and John F. Horty (1999). There's more to life than making plans: Plan management in dynamic, multi-agent environments. *AAAI Magazine* 20(4).

Rao, A. S. and M. P. Georgeff (1995). BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco.

Riedmiller, M. and T. Gabel (2007). On experiences in a complex and competitive gaming domain: Reinforcement learning meets robocup. In *Proceedings of the 3rd IEEE Symposium on Computational Intelligence and Games (CIG 2007)*, pp. 17–23. IEEE Press.

Röfer, Thomas (2005). Evolutionary gait-optimization using a fitness function based on proprioception. In *RoboCup 2004: Robot World Cup VIII*, Lecture Notes in Artificial Intelligence, pp. 310–322. Springer.

Russell, Stuart and Peter Norvig (2003). *Artificial Intelligence - A Modern Approach*. Prentice Hall, Upper Saddle River, New Jersey.

Schut, Martijn and Michael Wooldridge (2001). Principles of intention reconsideration. In *Proceedings of the fifth international conference on Autonomous agents*.

Singh, Satinder Pal (1992). Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning* 8(3–4).

Smart, W. (2002). Making Reinforcement Learning Work on Real Robots. Ph.D. diss., Department of Computer Science, Brown University.

Stone, P. and R. Sutton (2001). Scaling reinforcement learning toward RoboCup soccer. In *Proc. 18th International Conf. on Machine Learning*, pp. 537–544. Morgan Kaufmann, San Francisco, CA.

Stulp, Freek, Mark Pflüger, and Michael Beetz (2006). Feature space generation using equation discovery. In *Proceedings of the 29th German Conference on Artificial Intelligence (KI)*.

Takahashi, Y. and M. Asada (2001). Multi-controller fusion in multi-layered reinforcement learning. In *International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI2001)*, pp. 7–12.

Thrun, S. (2000). Towards programming tools for robots that integrate probabilistic computation and learning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, San Francisco, CA. IEEE.

Thrun, S., M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L.-E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, and P. Mahoney (2006). Stanley, the robot that won the DARPA grand challenge. *Journal of Field Robotics* .

Thrun, Sebastian (1992). The role of exploration in learning control. In White, David A. and Donald A. Sofge, editors, *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*. Van Nostrand Reinhold.

Thrun, Sebastian (1994). A lifelong learning perspective for mobile robot control. In *Proceedings of the IEEE/RSJ/GI Conference on Intelligent Robots and Systems*.

Thrun, Sebastian (1998a). Exploration in active learning. In Arbib, Michael, editor, *The handbook of brain theory and neural networks*, pp. 381–384. MIT Press, Cambridge, MA, USA.

Thrun, Sebastian (1998b). A framework for programming embedded systems: Initial design and results. Technical report CMU-CS-98-142, Carnegie Mellon University, Computer Science Department, Pittsburgh, PA.

Thrun, Sebastian B. and Tom M. Mitchell (1993). Lifelong robot learning. Technical report IAI-TR-93-7, University of Bonn.

Valsalam, Vinod, James Bednar, and Risto Miikkulainen (2006). Establishing an appropriate learning bias through development. In *Proceedings of the Fifth International Conference on Development and Learning*.

Vilalta, Ricardo and Youssef Drissi (2002). A perspective view and survey of meta-learning. *Artificial Intelligence Review* 2(18): 77–95.

Volpe, R. and S. Peters (2003). Rover technology development and infusion for the 2009 mars science laboratory mission. In *Proceedings of 7th International Symposium on Artificial Intelligence, Robotics, and Automation in Space (i-SAIRAS)*.

Weng, Juyang (2004). Developmental robotics: Theory and experiments. *International Journal of Humanoid Robotics* 1(2): 199–236.

Weng, Juyang, James McClelland, Alex Pentland, Olaf Sporns, Ida Stockman, Mriganka Sur, and Esther Thelen (2001). Autonomous mental development by robots and animals. *Science* 291.

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.

Williams, Brian C. and P. P. Nayak (1996a). Livingstone: Onboard model-based configuration and health management. In *Proceedings of AAAI-96*.

Williams, Brian C. and P. Pandurang Nayak (1996b). Immobile robots: Ai in the new millennium. *AI Magazine* 17(3): 16–35.

Witten, Ian H. and Eibe Frank (2005). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2nd edition.

Younger, A. S., S. Hochreiter, and P. R. Conwell (2001). Meta-learning with backpropagation. In *Proceedings of the International Joint Conference on Neural Networks (IEEE-2001)*.

Yu, Kai, Jinbo Bi, and Volker Tresp (2006). Active learning via transductive experimental design. In *Proceedings of International Conference on Machine Learning*.

Zell, Andreas et al. (1998). *SNNS User Manual*. University of Stuttgart and University of Tübingen. Version 4.2.