Robot Learning Language — Integrating Programming and Learning for Cognitive Systems

Alexandra Kirsch

Intelligent Autonomous Systems Group, Technische Universität München
Boltzmannstr. 3, D-85748 Garching
kirsch@in.tum.de

Abstract

One central property of cognitive systems is the ability to learn and to improve continually. We present a robot control language that combines programming and learning in order to make learning executable in the normal robot program. The language constructs of our learning language RoLL rely on the concept of hierarchical hybrid automata to enable a declarative, explicit specification of learning problems. Using the example of an autonomous household robot, we point out some instances where learning — and especially continued learning — makes the robot control program more cognitive.

Key words: robot learning, robot control language, hybrid automata, cognitive systems

1. Motivation

There are strong research efforts going on to develop complex technical systems for use in everyday life — car assistant systems, intelligent household devices, assistive systems for the elderly, mobile office applications — that are easy to use and react flexibly to changes in their requirements and environment. Such agents should learn constantly by observing their own behavior and adapting it to their requirements. We propose to consider learning as an integral part of the control program by special learning constructs in a programming language. This idea was described by Mitchell (2006) as one of the long-term goals in the development of machine learning:

> Can we design programming languages containing machine learning primitives? Can a new generation of computer programming lan-

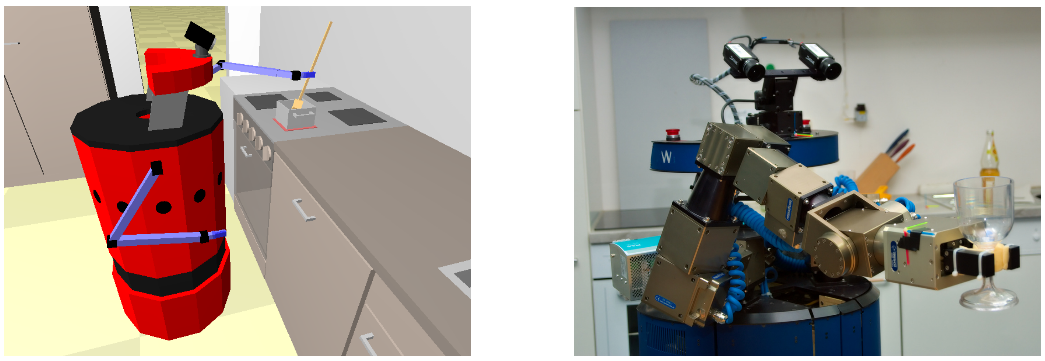


Figure 1: An autonomous household robot. We use a B21 robot in simulation (left) and the real world (right). The simulated arms are an imitation of the RX90 robot, whereas the real robot is equipped with two Powercube arms.

> guages 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.
> *Mitchell (2006)*

We consider an autonomous household robot as shown in Figure 1 as an interesting instance of a complex system that needs learning capabilities to adapt its behavior while doing its job. One challenge in implementing such a robot is that the outcome of actions in the real world strongly depends on lots of small parameters to be adjusted in an uncertain world like the direction from which to approach an object, where to stand in order to grasp it, the point at which to grasp it, etc. Additionally, a cognitive system needs prediction models for its own actions and how the world will evolve (with special focus on humans acting in the environment) and in the case of autonomous robots individual skills must also be learned and updated continually.

The sheer number of parameters, actions and models a robot needs suggests that they should be acquired automatically. Besides, dynamic environments require a robot to adapt to changing conditions by re-learning. In this article, we present the language RoLL (Robot Learning Language), which provides means to specify which part of the program is to be learned, how experiences are to be acquired, which ones are to be used for learning and which learning algorithm should be used. With RoLL, learning and programming can be combined to use the best solutions of both worlds.

Combining programming and learning — and thereby enabling continual improvement of the robot behavior — does not only speed up the development of some modules of a cognitive system. The modules and layers of abstraction of such a system depend strongly on other parts of the program. Therefore, when one part evolves, other program parts have new chances of enhancement as well. For example, when our robot improves its primitive navigation and grasping skills, this enables the robot to perform pick-and-place tasks more dexterously. However, before the pick-and-place tasks will work with the new underlying skills, the parameters of these tasks have to be adapted. Better pick-and-place skills enable the robot to perform more sophisticated activities. Instead of only setting the table, it can now start to learn how to load a dishwasher. Once the robot performs these new high-level activities, it experiences new situations, which will help to improve its low-level skills. Here the cycle starts again with the improvement of navigation and manipulation skills. In the course of this development, the robot has to adapt its models of its own skills constantly. For instance, before improving its low-level skills, the prediction model would have warned the robot that a certain grasping action will not succeed. This warning is superfluous when the skill has been enhanced.

This vision of a continually improving robot is the driving force of our research on the Robot Learning Language (RoLL). To our knowledge it is the first approach to integrate learning as a general concept into a programming language. We will point out the main challenges in implementing such a language and how we solved some of them. The next section gives an overview of RoLL's main concepts. Its declarative syntax heavily relies on the formal concept of hybrid automata. The relationship between hybrid automata and automatic learning is described in Section 3. After that, we present RoLL in more detail, first the automatic detection of experiences and then the learning operation. We then illustrate the declarative and elegant syntax of RoLL in a comprehensive example. We have evaluated the language with several learning problems, some of which are presented in Section 7. The article ends with a section on related work and a conclusion.

2. Overview

When learning is included into a control program, the program should still be easily readable and not show extensive traces of the learning mechanisms. RoLL offers declarative constructs for defining learning problems independently of the control program. Only two procedural commands were added for starting the experience acquisition and the learning process.

Figure 2 demonstrates how RoLL works. On the left-hand side it depicts the learning procedure in RoLL. On the right side of Figure 2 the declarative specifications are shown as pseudo code pieces 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 top-level control program could be the following:

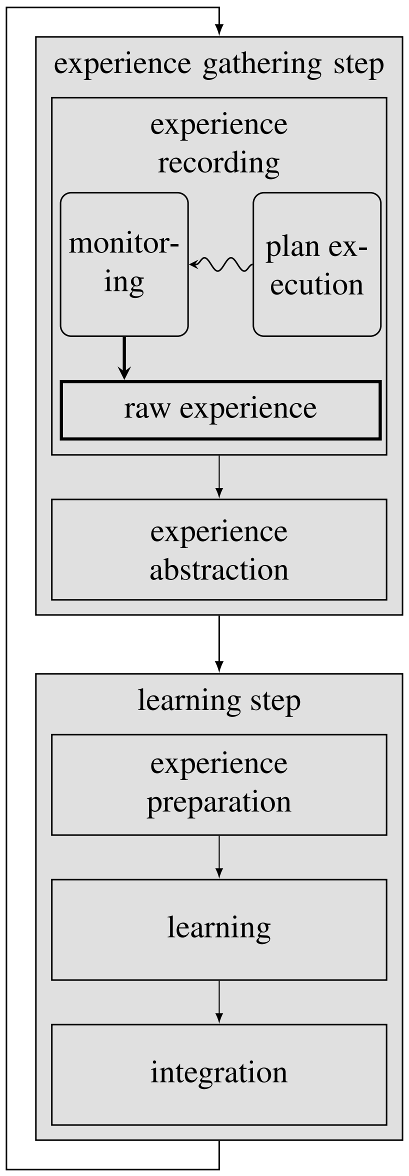
```
do-continuously
  do-in-parallel  acquire-experiences re
                  execute top-level-plan
  learn lp
```

This program executes some top-level plan, e.g. doing household work. In parallel it observes the experiences, which are specified declaratively in another part of the program. RoLL also supports the observation of several experiences in parallel. 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.

2.1. Experiences

One of the central concepts 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. We define experiences in terms of episodes in the following way:

> An *episode* is a stretch of time and all observable data available during this time. 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 necessary experiences for learning. State changes include events



```
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
```

Figure 2: Learning process in RoLL. On the right-hand side the corresponding code defining the learning process is shown.

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. We call the directly observed experiences *raw experience* and the converted ones *abstract experience*. 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 et al., 2005).

To implement this notion of experiences in a control language, our approach is based on the concept of hybrid automata for a theoretically grounded, well-understood underpinning of experiences and their impact throughout the learning process, which is explained in the next section.

The upper part of Figure 2 shows how experiences are acquired in RoLL. The programmer defines declaratively which parts (i.e. global or local variables and program states including failures) of the program are to be recorded as experience at which times. From this specification RoLL computes code that can be run in parallel to the normal program using `acquire-experiences` and writes the observed experience data to a data structure.

With a similar specification, the programmer defines the abstract experience — a feature language that makes the raw experience more suitable for learning. With this specification, an observed raw experience is converted directly to an abstract experience, which is stored in a database for offline learning or is used directly for online learning.

2.2. Learning

The second step — the learning step — starts with the last operation on experiences by transforming them to a format accepted by 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.

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

2.3. Implementation

Our learning language RoLL is implemented as an extension to the Reactive Plan Language (RPL) (McDermott, 1993, 1992). RPL is a concurrent reactive control language. It provides conditionals, loops, program variables, processes, and subroutines as well as high-level constructs (interrupts, monitors) for synchronizing parallel actions. To make plans reactive and robust, it incorporates sensing and monitoring actions, and reactions triggered by observed events. It makes success and failure situations explicit and enables the specification of how to handle failures and recover from them. RPL is implemented as an extension to LISP and therefore provides all its functionality beside the planning constructs.

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.

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 RPL task network and the information kept therein, the experience acquisition can take place completely independent from the rest of the program. With the concept of fluents changes in state variables are noted instantaneously by other processes.

3. Hybrid Automata as a Model for Robot Learning

All components of a learning problem are specified explicitly and declaratively in RoLL: 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 are comprehensible and universal so as to comprise all possibilities arising in arbitrary learning problems.

The biggest challenge was the definition schema of 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 comprise its execution status (i.e. which goals and routines are currently active or a failure state) and its internal state (i.e. all 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 using a hierarchical hybrid automaton. On the basis of this model of the program execution, the programmer specifies which experiences are to be observed at which moment in the modeled program.

3.1. 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.

In the context of 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.

The concept we need for experience acquisition are *hierarchical* hybrid automata. The example in the upper part of Figure 4 shows an automaton with two sub-automata that each contain another sub-automaton. The hierarchy allows the modeling of processes to an arbitrary depth as well as the option to describe processes at a very high level of abstraction.

3.2. Hybrid Automata for Robot Learning

Figure 3 illustrates the use of hybrid automata in the learning process. The first step in the learning procedure is the acquisition of raw experiences, which are then converted to abstract experiences. An example illustrating the steps of experience acquisition is shown in Figure 4. It shows the experiences for the learning problem of a function deciding which of its two grippers a kitchen robot should use for grasping an object.

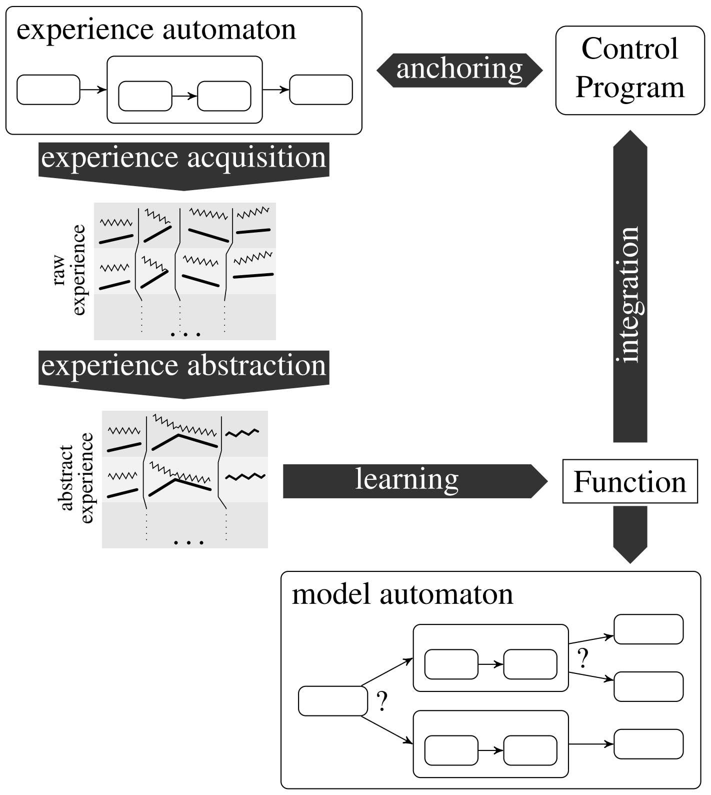
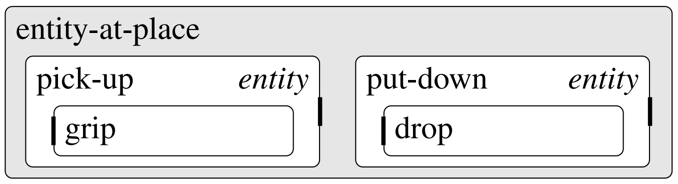
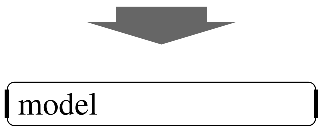


Figure 3: Use of hybrid automata in the learning process. We use hybrid automata as a means of modeling the control program and defining the desired raw and abstract experience. With the information of how this model relates to the real program, the specification is transformed to executable code. Learned models can also be considered as a partly specification of a hybrid automaton.

For defining a raw experience, an experience automaton is defined and anchored to the control program, which is executed in the environment (for example the automaton shown at the top of Figure 4). 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. The data can stem from external observations (global state variables) or from internal information about the program execution (active processes and local variables). In Figure 4 the angles of each robot joint are recorded while the goal *entity-at-place* is being achieved. Meanwhile, each time the goal *grip* is invoked, the poses of the entity to be gripped and the robot, the used arm and the current time are recorded. Other values are



entity-at-place$_{\text{interval}}$	→ *joint angles*
grip$_{\text{begin}}$	→ *entity pose, robot pose,used arm, timestep*
pick-up$_{\text{end}}$	→ *timestep*
drop$_{\text{begin}}$	→ *entity goal pose,timestep*
put-down$_{\text{end}}$	→ *entity pose, robot pose, timestep*



model$_{\text{begin}}$	→ *entity robot distance, handle orientation, used arm*
model$_{\text{end}}$	→ *time difference*

Figure 4: Experience abstraction for the learning problem of determining which hand to use for gripping. The assignment of data to an event (begin or end of automaton execution) is marked with vertical bars in the automata. Every experience consists of a hybrid automaton and data. The automaton structure is depicted above the data associated with it.

recorded at the beginning or end of the goals *pick-up*, *drop* and *put-down*.

The hierarchical nesting of hybrid automata provides a rich description for the observed data. In Figure 3 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 automaton 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 Figure 3, the hierarchical structure of the abstract experience is changed and internally and externally observed values are combined (indicated by wider zigzags).

The lower part of Figure 4 shows one way of abstracting the raw experience for the gripping problem. 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 some objective function for evaluating the gripping process. In Figure 4 this objective is the time needed. The value of the variables in the abstract automaton are defined on the basis of the ones in the raw experience, for example

$$\textit{time difference}@\text{model}_{\text{end}} = \textit{timestep}@\text{put-down}_{\text{end}} - \textit{timestep}@\text{grip}_{\text{begin}}.$$

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 3. 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 hierarchical 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. Here 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.

The possibility of modeling the program in the hierarchical automaton structure to arbitrary levels of detail provides a very flexible and convenient way for the programmer to declare experiences. If the modeling were restricted to an abstract level, some details of the program execution could not be observed. Contrarily, if the programmer were forced to model the program on the lowest level, experience definitions would be extremely hard to define and read.

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 and the whole learning process.

4. Experiences

This section and the next explain the RoLL language in detail. Code examples are presented in Section 6, which can also be used as a reference for the next two sections. In order to separate the learning code from the rest of the program, it is extremely important to have declarative constructs for defining the learning process. We present how RoLL achieves a high level of expressiveness without having to modify the original code of the robot program.

Experiences play a central role in the learning process. First, experiences must be observed while the robot is acting. After a useful observation has been made, the experience is abstracted and stored in the experience database.

We have defined experiences to be learning problem specific summaries of problem-solving episodes. This is mirrored in the data structure used for storing experiences in RoLL. The programmer specifies an episode by defining the structure of a hybrid automaton. The summary of information is filtered according to the definition of the desired data at special points in the automaton.

4.1. Raw Experience Detection

Raw experiences are the direct observations of beliefs and internal robot parameters during execution.

The robot is controlled by some program, which can either be the robot's standard control program or one specially designed for experience acquisition. An independent monitoring process observes internal and external parameters that are changed by the control program. It records relevant data and passes the whole experience on for abstracting or using it. 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.

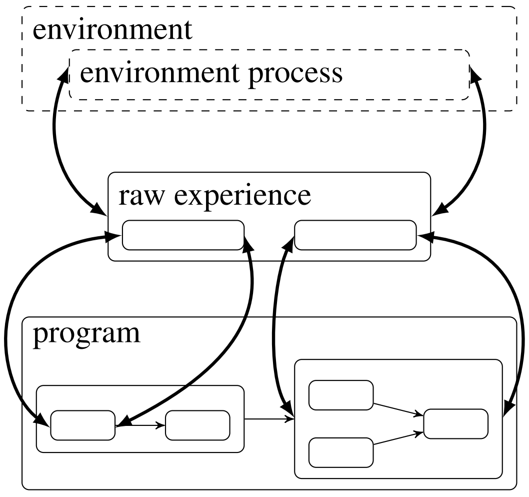


Figure 5: Description of the HHA structure for raw experience acquisition. The arrows indicate how the experience automaton can be anchored in the automaton defined by the robot program and in the environment process.

4.1.1. Experience Automaton and Anchoring

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. For an example of such a definition see Listing 6.1.

The automaton structure itself is described by a hierarchy of subautomata. as shown in Figure 4 on page 10. 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 subautomaton.

For identifying interesting episodes in the program execution, the automaton specification of the experience must be associated with events inside and outside the robot program during its execution. We call this process “anchoring”.

There are two sources of anchoring as illustrated by Figure 5: events in the environment and events inside the program. We have shown in Section 3 how the control program can be modeled in the framework of hybrid automata. This means that we can associate a subautomaton of the program with the specified experience automaton. To do this, we 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 goal to be achieved, the execution of a routine, or the execution of any task that has been marked by a global tag. After addressing an RPL task via the goal, routine or a global tag, all other program parts are accessible by navigating through the task tree. For most purposes, the standard introductory points are sufficient, like observing each time a certain goal is to be achieved. For acquiring experiences that need more detail of the program, for example the duration of the first step in a plan execution, the specification of the experience relies strongly on the actual program structure. When the program is changed, the experience definition might have to be changed too. A more robust alternative is the definition of a tag to mark the interesting part of the control program.

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 automaton is described by the invariant condition of someone being inside the kitchen. In the code this condition is given by a fluent variable[1], which is always true when a human is in the room and switches to false when no person is present.

In sum, 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 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.

4.1.2. Experience Data

Having defined the episode in the form of a hybrid automaton, we now have to add the specification of the desired data points. Variable values can either be stored once at the beginning or end of the automaton execution or continuously

[1]A fluent in RPL is a variable whose changes are notified automatically to other processes. For an invariant the fluent must represent a Boolean value.

during its activation. For the events and the interval there can be different sets of variables to be recorded.

The values to be recorded can stem from global variables like state variables, which are accessible from all over the program (e.g. the robot's position). 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 or the arm it uses for a manipulation task. 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 experience automaton to the program.

The question of which data should be recorded in an experience lies with the programmer and depends on the learning problem(s). In general it is a good idea to observe more aspects of a problem than seems to be necessary for a given learning problem. In most cases, the outcome of a learning problem relies strongly on the abstract feature language that is used for the learning process. With more data, it is easier to modify the later steps of experience abstraction without the need to observe new experience. Besides, it can be reasonable to acquire raw experience that is to be used for several learning problems. For example, when a routine needs models about the time needed to achieve the goal and about its success rate, the necessary data is almost identical. In this case it makes sense to observe only one raw experience with the information for both learning problems and later generate two abstract experiences out of it.

4.1.3. Failures

The execution of a robot plan can always fail. This may or may not affect a specific experience acquisition process, it might even be an event to observe as an experience.

There are two kinds of failures that can happen during experience acquisition: a plan failure leading to the current plan being stopped or a condition that is not recognized as a failure in the program, but makes the currently observed experience uninteresting. Failures in the control program are detected in the main program code and are either handled to recover from the failure or lead to the interruption of the plan. Undesirable conditions for a specific experience can be given in the experience specification.

No matter how the failure was detected, the programmer decides for each experience what should happen to the possibly incomplete data that has been acquired for the failed episode. Possible reactions include discarding the data or using it nevertheless. In the latter case, the data can be modified before being

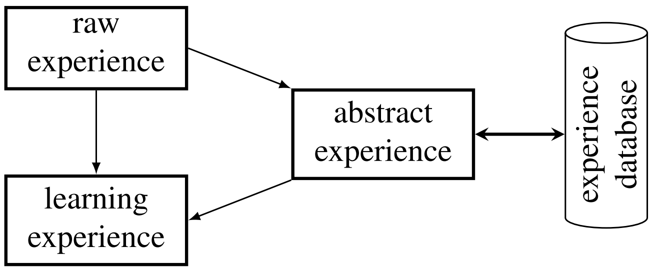


Figure 6: Typical experience abstraction steps. The raw experience is abstracted to an intermediate step, which is stored permanently. This experience can be retrieved from the experience database for different learning problems and be abstracted again for learning.

stored as an experience by adding, deleting or replacing parts of the experience data. For deciding on an appropriate reaction, RoLL offers information about which data points have already been stored before the failure occurred, thus allowing conditional reactions.

4.2. 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. 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, because 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 6: The first conversion is performed directly after the observation, its result 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.

Experiences can be stored in different ways, for example in an array, a log file or a database. RoLL has the concept of *experience classes*, which hide the implementation of the storage medium from the programmer who specifies an experience. An experience class provides an interface for storing and retrieving experiences in a uniform way.

An abstract experience thus contains (1) a raw experience, whose data is used as input, (2) a hybrid automaton specifying the data transformation and (3) the experience class. The syntax for specifying an abstract experience is very similar to that of a raw experience, also relying on the concept of hybrid automata. The language is general enough to allow all algebraic and conditional translations between experience data. An example is shown in Listing 6.2.

4.3. Example

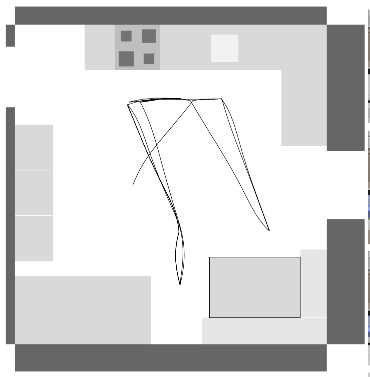
To give a flavor of the things that can be observed with the experience acquisition mechanisms in RoLL, Figure 7 shows an execution trace of a plan to prepare pasta. This example was used for monitoring and assessing a plan, but the data could also be used for learning prediction models of each subplan. The raw experience models the execution of the preparing pasta plan by including subautomata for frequently used plans like navigation, gripping and putting down an object. For each plan it observes generic data such as the duration of the plan, how often it was invoked during the pasta preparation plan, the robot's position, parameters of the plan execution like the arm the robot chose, and failures.

The raw experience can be used to provide information about the plan execution. The upper left picture of Figure 7 depicts the way the robot has moved during the plan execution. For evaluating the quality of the plan, we calculated some statistics like the overall time and the duration of each subplan, the distance moved and how far the robot turned in the whole plan.

The same values could be used for learning prediction models about the duration of each subplan or the expected space the robot will be occupying for a given task. This experience definition is very complex and extensive so that it would provide abstract experiences for a wide variety of learning problems. For the learning problems we describe in Section 7, we used smaller experience definitions, which only observe specific aspects of the control program that were interesting for each learning problem.

5. Learning Programs

We have described in detail the RoLL language constructs concerned with experiences. Now we explain the learning part. The specification of learning problems is declarative, like that of experiences and the declarations are then trans-



	duration	count	sum
plan	1138.10 s		
navigation	195.75 s	17	21.96 m
turning	52.21 s	15	820.55°
left arm	145.39 s	79	17.44 m
right arm	191.41 s	99	23.29 m

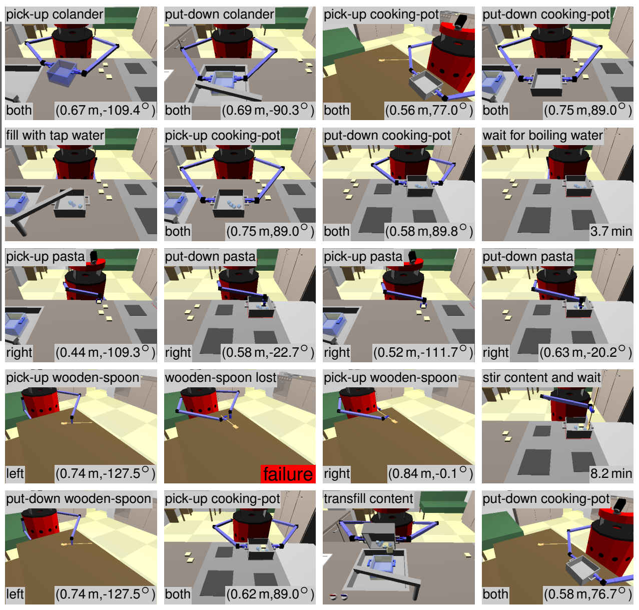


Figure 7: Trace of an experience acquisition process. The experience includes the current action (in the left corner of each small image), the robot's position (bottom right corner) and its decision on which hand it uses (bottom left corner). On the left, some abstractions from these observations are shown: the ways the robot moved in the world (picture on top) and some statistical measures from the plan observation (table on the bottom).

formed to executable code automatically. The main components of a learning problem — apart from the experiences to be used — are the type and identity of the function to be learned and the learning algorithm. An example for a learning problem definition is shown in Listing 6.3.

5.1. Learning Problem Classes

A robot control program includes a variety of function types to be learned, e.g. prediction models, parameterization functions, decision rules, control routines, 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 prediction model must be added to the program in a different way than a routine (in the latter case, the function must be embedded into a control loop, for instance).

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 (the parameters they take and the result they return) and the way the resulting function is integrated into the control program.

5.2. Learning Algorithms

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 et al., 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 uses a specific class of experiences. We have mentioned experiences that are stored in databases. RoLL can be extended with arbitrary experience classes that store their data in different formats. For a learning system, this means that the experience class stores the learning data in a way that the learning system can use it.

A learning system may assume a certain structure of the abstract experience automaton, for example a neural network learner may require a flat automaton without subautomata and may assume that the data representing the beginning of the automaton execution contains the input values of the network and the data associated with the end of the automaton run represents the output value.

Beside the experience type, a learning system in RoLL must provide the call to an external program or 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.

5.3. Integration of Results

Beside the main components of a learning problem — experiences, function to be learned and learning algorithm — there is another tricky part to be specified for enabling the integration of the learning result as a callable function into the program.

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 time 1 and the commands given at time 0. After observing

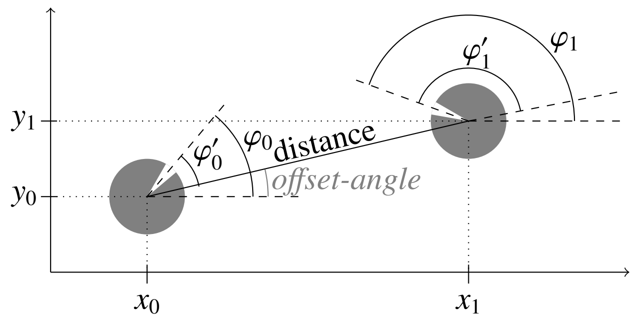


Figure 8: Illustration of experience abstraction for learning a navigation routine. The original state space $\langle x_0, y_0, \varphi_0, x_1, y_1, \varphi_1 \rangle$ is transformed to $\langle distance, \varphi_0', \varphi_1' \rangle$.

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 $\langle rot_0, trans_0 \rangle$.

For learning, we define a feature space $\langle distance, \varphi_0', \varphi_1', rot_0, trans_0 \rangle$ that is invariant with respect to rotations and shifts in 2D space. The correlation of the original experiences and the abstract feature space is depicted in Figure 8. Now the signature of the learned function is $distance \times \varphi_0' \times \varphi_1' \to rot_0 \times trans_0$. This is not the only possible state space representation and the decision which features to use for learning is part of the learning problem specification, not the function to be learned. This resulting function will be called in a straightforward way giving the goal position $\langle x, y, \varphi \rangle$ and expecting a rotational and translational velocity command without caring about the abstractions performed for the learning process.

Figure 9 illustrates this phenomenon. The function F is the one that is intended to be learned with the signature $x \times y \times z \to v \times w$, whereas f is the function produced by the learning algorithm with the signature $h \times i \to j \times k$. 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?

To illustrate the first question, consider again the example of the navigation

$$\begin{tikzcd}
\langle x, y, z\rangle \arrow[r] \arrow[d, "A_0"] & \boxed{\mathrm{F}} \arrow[r] & \langle v, w\rangle \\
\langle a, b, c\rangle \arrow[d, "A_1"] & & \langle d\rangle \arrow[u, "A_0^{-1}"] \\
\langle e, f\rangle \arrow[d, "A_2"] & & \langle g\rangle \arrow[u, "A_1^{-1}"] \\
\langle h, i\rangle \arrow[r] & \boxed{\mathrm{f}} \arrow[r] & \langle j, k\rangle \arrow[u, "A_2^{-1}"] \arrow[uuu, dashed, bend right]
\end{tikzcd}$$

Figure 9: Abstraction in the context of the whole learning process.

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.

For an automatic back transformation that converts $\langle j, k\rangle$ to $\langle v, w\rangle$ (in Figure 9 indicated by the dashed arrow) it would be necessary to invert functions. There is no straightforward way to do this in a general case — for example, when a LISP function is used that includes conditionals and recursion to produce a result. Therefore, in RoLL the back transformation has to be specified manually by the programmer. Although this specification requires some repeated work by the programmer (in the state space specification for learning and in the integration part of the learning problem definition), this solution avoids unnecessarily complicated syntax for difficult cases in the back transformation. Besides, the transformation of the output value is usually very simple and straightforward, so that the extra specification doesn't require a lot of work.

6. Example

To illustrate the RoLL language we present the learning problem of a time prediction model for a navigation routine, which is called `go2pose-pid-player`. It takes the robot from its current position to a goal position, which is specified by 2D coordinates and an orientation. We learn a mapping to the time needed to fulfill a navigation task using regression trees as our learning system. We present the learning process according to the steps in Figure 2 on page 5.

6.1. Raw Experience Acquisition

The first step is the observation of raw experiences. The definition of the raw experiences for this problem are shown in Listing 6.1. The experience automaton consists of one automaton without children and is anchored to the control program by defining the activity of the routine `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 local routine description, which is bound in the lexical scope of the RPL task corresponding to the navigation routine (line 7, 8). When the navigation task is aborted or has failed, the data is discarded (lines 10, 11).

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

Listing 6.1: Raw experience definition for a time prediction model of the navigation routine `go2pose-pid-player`. Each time this routine is active, the start and goal poses are recorded as well as the time at the start and end of the execution. The data of incomplete runs is discarded.

For observing these experiences, we let the robot perform its usual duties. In our case, this is 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)))
```

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

Listing 6.2: Abstract experience and conversion specification for the navigation time model. Using intermediate variables as shown in Figure 8, the resulting feature space is *dist*$\times$*start-phi*$\times$*end-phi* $\rightarrow$ *navigation-time*. The experience data is to be stored in a database.

The `acquire-experiences` command can be used anywhere in the program. However, one motivation for using RoLL is not to modify the program for the purpose of experience acquisition. 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.

6.2. 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 shifted or rotated on the 2D plane are treated as different cases, but should return the same result. Therefore, we use the abstraction depicted in Figure 8 on page 20, 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). It contains successive variable declarations, corresponding in syntax and functionality to 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 offset angle indicated in Figure 8. Based on these values, the distance of the two points and the normalized orientations are calculated (lines 10–13).

We store the abstracted experience in a database, which must be specified in the experience (lines 17–18). The access to the database is performed by the experience class `database-experience` and is hidden from the programmer of the abstract experience specification. Likewise, the data is retrieved automatically.

6.3. 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 `go2pose-pid-player` (line 2), which is a function that is associated to the routine as a predictor for the expected duration. From this specification, RoLL automatically assigns this learning problem the unique identifier `go2pose-pid-player-time-model`. 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 19–20), we adjust the algorithm to learn a regression tree instead of a model tree (line 21).

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).

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

```
(roll:define-learning-problem
 :function (:model go2pose-pid-player :time)
 :use-experience
  (:parent-experience navigation-time-abstract-exp
   :specification
    (:anonymous-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 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 *root-dir* '("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))
```

Listing 6.3: Learning problem definition for the navigation time model. The function to be learned is the time model of the routine `go2pose-pid-player` using the experience as defined in Listing 6.2. As a learning algorithm it uses the M5' algorithm for regression trees.

```
(roll:learn
  (getgv :learning-problem 'go2pose-pid-player-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 can for instance be used 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.

Currently all parts of the learning process — experiences and the learning problem definition — have to be specified by the programmer. But having such an explicit representation paves the way to generate some of the specifications. Vilalta and Drissi (2002) provide a survey of meta-learning algorithms, which could be used as a starting point to automate the choice of the bias or the learning system. Another interesting question is how to generate a good state space abstraction, which has been studied to some extend by Herrera et al. (2006) and Stulp et al. (2006), for instance. In further steps, the robot itself should be able to decide which parts of its program it wants to (re-)learn and when. In Section 7 we describe a method to execute two activities in parallel. This interleaving of actions relies on learned predictions about the duration of each action. The learning problems for these prediction models were generated after a common pattern for several robot actions.

7. Evaluation

The strength of RoLL lies in observing experiences for and learning lots of small learning problems at the same time. This enables the robot to evolve over time. In the following we describe some problems we have learned with RoLL and then demonstrate how learned models can be used to optimize the robot's overall behavior. None of the problems in itself is particularly difficult to solve. That's why we don't explain them to great detail. The important point is to show how the robot can improve its overall behavior by learning and adapting continually. The learning problems described in the following are summarized in Table 1.

For enabling cognitive behavior, a robot needs expectations about the world and must make predictions about its own plans. To do so, we implemented learning problems for learning prediction models such as the one for the navigation task explained in Section 6. We also learned prediction models for single grasping actions and more complex pick-and-place activities. With such prediction models, the robot can monitor its own behavior and detect errors that are outside of its control like hardware failures. We used the model of the navigation routine to define a timeout condition for navigation tasks depending on the specific task in a specific situation. When the predicted time is exceeded significantly, the robot aborts the navigation task with a failure, because it can assume that something went wrong, although it cannot observe what it was.

For manipulation activities, two important questions for our robot are "Where should I stand?" and "Which arm should I use?". For choosing an arm for picking

Learning Problem	Experience	Learning Algorithm	Use of Result
	Prediction Models — Duration of Actions		
navigation	current and goal pose, time stamp at start and end	regression tree	timeout condition for failure recognition
grasping, complete pick-and-place task	original and goal position of object, original (and goal) position of robot, used arm, timestamp and start and end	neural network	decision which arm to use
10 different subtasks of preparing pasta and setting the table (e.g. time to put the pot on the cooker)	start and end time of each task	average value	parallel execution of activities
	Routines — Navigation with Bézier Curves		
parameters of Bézier curve	start and goal position, used curve parameters, duration, success	neural network	navigation in the RoboCup domain
following waypoints	start and goal position, given command for rotational and translational velocity, duration, success	neural network	following a Bézier curve for navigation
	"Meta-Learning" (search was performed manually)		
time prediction for navigation	current and goal pose, time stamp at start and end	neural network, regression tree, model tree	choice of learning system for this specific problem

Table 1: Overview of learning problems solved with RoLL.

up (and later putting down) a cup, we used prediction models for picking up a cup and putting it down at its goal position for both arms. At run time, the robot compares the predicted time needed with the left and the right arm and chooses the more efficient one.

Instead of single parameters, it is also possible to learn complete control routines with RoLL. Our robot learned how to navigate by using two learning problems: a high-level problem for determining the parameters of a Bézier curve the robot is to follow and a low-level problem for following that Bézier curve (Kirsch et al., 2005). In this context we experimented with choosing a good set of experiences for learning from all the acquired experiences.

Because RoLL makes all the components of a learning problem explicit, it is an ideal framework for meta-learning. As meta-learning is not our research focus, we didn't implement a complete meta-learning cycle. However, when learning the prediction model of the navigation routine, we used RoLL's abilities to use several learning algorithms for comparing different algorithms and different parameterizations. In several iterations, we learned the prediction model with the same experience using different learning algorithms (model tree, regression tree and neural network). We then compared the outcome by again collecting experiences, but using them as a test set instead of learning data. In this specific case, regression trees had the lowest error rates.

More interesting than single learning problems is the use of learned models in the context of parallel plan execution. Our robot has a plan for setting the table and one for preparing pasta. Especially the latter task leaves unused time that the robot could use to set the table. For interleaving the execution of the two plans, the robot needs predictions of how long plan steps will take, and 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 data about the duration of the action. The learning consisted in a simple averaging of the durations observed for all the subtasks. Then the predicted times were used to transform the sequential execution of two plans into a parallel execution. With this transformation instead of 907 s when executing the plans sequentially, the robot needed only 708 s for both plans, which is only 12 s more than for the pasta plan alone.

Learning is not the only necessary component to make a cognitive robot program. We are also exploring transformational planning techniques for improving the robot's behavior. In this context, we also used the experience acquisition capabilities in RoLL to evaluate plans and observe their execution in order to find appropriate transformation rules for improving them.

In all, these examples show that RoLL is very versatile. It can be extended with learning algorithms, by defining new types of learning problems and by storing experiences in different ways. Because of its declarative nature, it is even possible to generate learning problems such as the prediction models needed for executing the two high-level plans in parallel. By learning more and more parts of the program and integrating the learned parts with programmed ones, systems can adapt dynamically to their environment and constantly enhance their abilities.

8. Related Work

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, 1998).

Andre and Russell (2001) 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 issue of experience acquisition gets 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 agent's 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 and Russell (2001). The focus of IBAL is not on learning in general, but on programming with probabilistic models. 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. (2000) also emphasize that the best results can be obtained by a smooth interaction of programming and learning compared to learning or programming alone.

The idea of constantly enhancing the robot's knowledge also occurs in the work of Schultz et al. (1999). They use a common representation of the environment in the form of evidence grid maps for robot exploration, localization, navigation and planning. These maps are updated each time new evidence is observed, thus enabling the robot to recognize changes in the environment and to build very accurate maps. In contrast to RoLL, the approach is restricted to map learning.

Beside the learning capabilities, RoLL's constructs for specifying and acquiring experiences from the program execution are very useful for self-inspection and plan transformation. The acquisition of data has been studied in the context of monitoring and debugging. In the XAVIER project (O'Sullivan et al., 1997) of Carnegie Mellon University all available input and output to the robot 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.

The Common Lisp Instrumentation Package (CLIP) (Anderson et al., 1994) is a Lisp package for inspecting and testing programs. Its goal is to provide a standardized framework for data collection, where the functionality of the program is clearly separated from the data acquisition part. This separation, however, only goes as far as the 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.

9. Conclusions

The central characteristic of a cognitive system is its ability to evolve and adapt to unknown situations. Although learning has been used to enhance the capabilities of autonomous robots, it has mostly been used for function optimization in the design phase.

We have presented the robot control language RoLL that includes learning capabilities on the level of programming to enable experience acquisition and continual adaptation during the robot's operation. RoLL proposes a general framework for all kinds of experience-based learning methods, which allows to use arbitrary learning algorithms. It provides declarative programming constructs for the whole learning process. In particular, we have presented a declarative way of defining experiences for learning based on the concept of hybrid automata, allowing the acquisition of training data without modifying the main robot program. We have also pointed out the difficulties in integrating the learned function into the program independent of the feature space defined for learning.

Currently, RoLL doesn't make learning less complex. The design of a learning problem with RoLL involves the same decisions as in the conventional way. But RoLL makes the learning process executable and allows to repeat it during the operation of the robot. Even though the learning problem definition is fixed, the behavior of the robot will improve with more experiences and adapt to changes in the environment.

Capturing the whole learning process in explicit specifications also helps in the engineering process of a learning problem. RoLL allows to change the set of experiences, the feature space and the learning algorithm without much effort. The experience acquisition also allows to monitor the performance of different learned behaviors, which enables the comparison of different definitions of a learning problem. Similarly, RoLL can serve as a tool to develop and compare different learning algorithms for autonomous robots.

Once one has a system such as RoLL, a lot of new research questions come up for making the learning process more automatic. The specifications that currently have to be provided by the programmer could be generated automatically with meta-learning algorithms and automatic feature extraction methods. Moreover, a robot could decide which skills it needs to improve and when, which involves the well-known exploration-exploitation problem that has been investigated in the area of reinforcement learning. Finally, a robot could even define new learning problems when it finds that it needs more models or has to improve a certain skill. These research questions only get interesting when learning is defined as an executable process and RoLL provides a means to start tackling them.

In sum, the thorough integration of learning into robot control programs is an important step towards cognitive systems. On the one hand, it allows to adapt the control program automatically to changes in the environment. On the other hand, it provides an efficient way to acquire and update models of the environment, which the robot needs to make sound decisions in a the physical world. Besides, the explicit specification of learning problems allows to develop new methods to generate some of the definitions.

We believe that the embedding of learning capabilities into a control language in the way it is implemented in RoLL is a necessary prerequisite in developing cognitive systems. Such a language doesn't solve all the problems of self-awareness and adaptation in a cognitive system, but it is an essential step for the development of these methods.

References

Alur, R., Belta, C., Ivancic, F., Kumar, V., Mintz, M., Pappas, G., Rubin, H., Schug, J., 2001. Hybrid modeling and simulation of biomolecular networks. In: Fourth International Workshop on Hybrid Systems: Computation and Control. pp. 19–32.

Anderson, S. D., Westbrook, D. L., Hart, D. M., Cohen, P. R., January 1994. Common Lisp Interface Package CLIP.

Andre, D., Russell, S., 2001. Programmable reinforcement learning agents. In: Proceedings of the 13th Conference on Neural Information Processing Systems. MIT Press, Cambridge, MA, pp. 1019–1025.

Boutilier, C., Reiter, R., Soutchanski, M., Thrun, S., 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.

Branicky, M. S., 1995. Studies in hybrid systems: Modeling, analysis, and control. Ph.D. thesis, Massachusetts Institute of Technolgy.

Henzinger, T., 1996. The theory of hybrid automata. In: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96). New Brunswick, New Jersey, pp. 278–292.

Herrera, L. J., Pomares, H., Rojas, I., Verleysen, M., Guilén, A., 2006. Effective input variable selection for function approximation. In: International Conference on Artificial Neural Networks. Lecture Notes in Computer Science. Springer, pp. 41–50.

Kirsch, A., Schweitzer, M., Beetz, M., 2005. Making robot learning controllable: A case study in robot navigation. In: Proceedings of the ICAPS Workshop on Plan Execution: A Reality Check.

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

McDermott, D., 1993. A reactive plan language. Tech. rep., Yale University, Computer Science Dept.

Mitchell, T. M., July 2006. The discipline of machine learning. Tech. Rep. CMU-ML-06-108, Carnegie Mellon University.

O'Sullivan, J., Haigh, K. Z., Armstrong, G. D., April 1997. Xavier.

Pfeffer, A., 2001. IBAL: A probabilistic rational programming language. In: IJCAI. pp. 733–740.
URL `citeseer.nj.nec.com/447186.html`

Schultz, A. C., Adams, W., Yamauchi, B., 1999. Integrating exploration, localization, navigation and planning with a common representation. Autonomous Robots 6 (3), 293–308.

Stulp, F., Pflüger, M., Beetz, M., 2006. Feature space generation using equation discovery. In: Proceedings of the 29th German Conference on Artificial Intelligence (KI).

Thrun, S., 1998. A framework for programming embedded systems: Initial design and results. Tech. Rep. CMU-CS-98-142, Carnegie Mellon University, Computer Science Department, Pittsburgh, PA.

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). IEEE, San Francisco, CA.

Vilalta, R., Drissi, Y., 2002. A perspective view and survey of meta-learning. Artificial Intelligence Review 2 (18), 77–95.

Witten, I. H., Frank, E., 2005. Data Mining: Practical machine learning tools and techniques, 2nd Edition. Morgan Kaufmann, San Francisco.

Zell, A., et al., 1998. SNNS User Manual. University of Stuttgart and University of Tübingen, version 4.2.