A relational approach to tool-use learning in robots

Solly Brown

Submitted in fulfilment of the requirements for the degree of Doctor of Philosophy, September 2009.

School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

Originality Statement

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgment is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Solly Brown

Copyright Statement

Solly Brown

Authenticity Statement

'I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.'

Solly Brown

Abstract

This thesis presents a robot agent which learns to exploit objects in its environment as tools, allowing it to solve problems which would otherwise be impossible to achieve. Our agent learns by watching a single demonstration of tool use by a teacher, and then by experimenting in the world with a variety of available tools. The emphasis in our approach is on learning tool-use in a relational context, and our agent is able to generalise across objects and tasks to learn the spatial and structural constraints which describe useful tools and how they should be employed.

Two learning mechanisms are employed to achieve this: learning by explanation, and learning by trial-and-error. A form of explanation-based learning is used to identify the most important sub-goals the teacher was able to achieve by using the tool. The action model constructed via this explanation is then refined through trial-and-error experimentation and the use of a novel Inductive Logic Programming (ILP) algorithm.

Acknowledgements

I'd firstly like to thank my supervisor, Claude Sammut, for his advice and guidance during the course of this PhD. Claude encouraged me to explore a new and interesting area in this research, and has always been supportive of my work. Thank-you Claude!

I would also like to acknowledge the assistance of Morri Pagnucco, Bernhard Hengst, Mike Bain, Will Uther, and Alan Blair, who have all at one time or another provided me with helpful and insightful advice. Morri was also good enough to proof-read large sections of my thesis, and help me out in his role as post-graduate coordinator.

This research has been supported financially through an Australian Postgraduate Award, National ICT Australia (NICTA) Postgraduate Award, and Centre for Autonomous Systems (CAS) scholarship. The support of the Australian government, along with both CAS and NICTA, is gratefully acknowledged.

I've enjoyed my time spent in the robotics lab at CSE and would like to tip my hat to everyone who has made it a fun place to work. Thanks to Martin de Groot, Cameron Stone, Min Sub Kim, James Wong, David Johnson, Raymond Sheh, Nawid Jamali and Anna Wong (apologies to anyone I've forgotten!). A special mention to John Zaitseff for always being willing to help with the dark arts of linux systems admin!

Lastly, the biggest thank-you must go to my family and friends for supporting me through the many ups and downs of this PhD. Mum, who's no longer with us, would have been delighted to see me finished. Dad, thanks for your unwavering support and for staying up till 3am the night before I was due to fly out of the country, to help me print out this thesis! Jess, I'm really sorry I took so long to finish so thank you for your love, patience and understanding — and I promise never to do a PhD ever again!

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

In this thesis we address the problem of tool use and learning in robots. Robots, humans and animals have a limited range of effectors and can use tools to extend the range of problems they can solve. A household robot for example, may need to use a chair to prop open a door, use an old rag to soak up a puddle from a leaky fridge, or simply use a tray to carry a collection of glasses from one room to another.

Our goal is to develop algorithms for a robotic agent which can learn to use objects in its environment as tools, in order to solve problems which would otherwise be more difficult or impossible to achieve. Our robot agent begins with rudimentary knowledge of the objects in its world and learns through demonstration and experimentation that some objects in its world have useful properties which can be exploited for solving problems.

1.1 Tool use in animals

Contrary to popular belief, tool use is not a uniquely human trait. Tool use has been observed in a wide range of both captive and wild animals: chimpanzees use sticks to extract termites from termite mounds (van Lawick-Goodall, 1970); Egyptian vultures select appropriately sized stones to crack open stolen eggs (Goodall and van Lawick, 1966); burrowing owls use mammalian dung to 'fish' for dung beetles (Levey et al., 2004); and some bottle-nosed dolphins use sponges on their

noses to protect themselves from stings whilst foraging on the sea floor (Krützen et al., 2005). Some creatures such as the New Caledonian crow are even adept at tool-making and solving novel problems using tools (Hunt, 1996; Kenward et al., 2005).

Animals that make use of tools do so in order to enable or improve their ability to carry out important tasks. The New Caledonian crow, for example, uses twig and leaf-based hook tools to extract grubs from crevices — a feat that would be more difficult or impossible without these tools. By exploiting objects in the environment they are able to overcome the limitations of their effectors and expand the range of goals they can achieve.

Humans are, of course, more proficient at using tools for problem solving than other animals. No other creature is able to exploit objects in its environment more effectively, or shows as wide-ranging an ability to use tools, as humans. Tools are so ubiquitous in our everyday lives that there is scarcely an activity we perform which does not involve using one or more objects to make the task possible, or to enable it to be carried out in a simpler or more efficient manner. On my desk for example, sits a collection of pens, a pair of scissors, a computer, a glue-stick, a bike helmet, a telephone and a considerable quantity of paper. In fact, almost every object in my immediate vicinity is a tool of some sort.

1.2 A tool-use learning robot

Given the ubiquity of tools in our man-made environment we would like (in the future) to have robots which can take advantage of them. It would be useful, for example, if a household robot was able to learn to use new tools. Whilst one might expect that a household robot would come pre-programmed from the factory with many useful behaviours, there would remain many objects or situations which the programmers had not anticipated — for example, using a pot plant to prop open a door, or using a spoon to pry the lid off a tin of cocoa.

We would like a tool-using robot to be able to learn to use a new tool after observing it being used correctly by another agent. A robot might observe someone using a spoon to open the cocoa tin, and notice that they have managed to achieve a

useful goal which was not previously considered achievable. The robot might then wish to learn this useful action by attempting to open a tin itself. This would involve selecting a suitable object to use as the tool, and then trying repeatedly to use it in the correct manner to remove the lid. If the robot selected the wrong type of spoon (eg. plastic, too small, or too thick) for the given tin then it would need to experiment and try to identify one which works.

There are four important things which the robot needs to learn in order to consistently solve the tool-use task correctly:

1. What useful effect the tool action achieves
2. How to identify a good tool — what object properties are necessary?
3. The correct spatial position in which the tool should be placed
4. How the tool should be manipulated after being placed

In this thesis we focus on learning the first three of these and assume that the fourth – manipulation – is simple enough that it can be achieved via a primitive action (eg. pulling downwards on the spoon once it is position) or an action which can be generated by a motion-planner (such as placing the pot plant against the door). More complex manipulation is beyond the scope of this thesis.

The general case of the learning problem we address in this thesis can be stated as follows. The learner agent observes a "teacher" agent using a tool to solve a task and achieve a useful goal. We will refer to the teacher's example as the *demonstration*, and assume that only a single demonstration is available to the learner. The agent is then presented with a series of new learning tasks, each of which involves a variety of different available tools. By analysing the teacher's demonstration and then experimenting directly in the world the aim is to learn a new tool action which will allow the agent to perform new versions of the task.

The agent's progress on learning to use the tool can be evaluated by measuring how successfully it can perform new versions of the same task. As the agent learns to better identify tools that can be used successfully (by isolating the important properties), it will require fewer attempts to perform a given instance of the particular task. Similarly, the agent's success rate will increase when it learns to generalise the correct spatial positioning of the tool across different situations.

In Section 1.4 we present a concrete example of a tool learning task tackled in this thesis, and in Section 1.5 describe our approach to learning to solve tool-use problems.

1.3 What makes an object a tool?

It is useful to briefly define exactly what we mean by tools and tool use, since there is no universally accepted definition. Celebrated anthropologist Jane Goodall described tool use as (van Lawick-Goodall, 1970)

> *"the use of an external object as a functional extension of mouth or beak, hand or claw, in the attainment of an immediate goal."*

While this definition is perfectly reasonable, it is a narrow one which rules out objects like paperweights and ladders being considered as tools. With this in mind most researchers now accept a more general, if somewhat less concise, definition of tool use provided by Benjamin Beck (Beck, 1980):

> *"[Tool use is] the external employment of an unattached environmental object to alter more efficiently the form, position, or condition of another object, another organism, or the user itself when the user holds or carries the tool during or just prior to use and is responsible for the proper and effective orientation of the tool."*

Beck's definition covers most interesting cases of tool use that one might wish to consider. Under this definition a paperweight is a tool because it can be used to weigh down pieces of paper and stop them blowing away. A ladder is a tool because it can be used to alter the vertical position of the user in the world.

1.3.1 The definition of tool-use used in this thesis

In this research we are less concerned with the semantics of tools and tool use, and more with what tools can help an agent achieve. We define a tool as follows:

> *An object is considered a tool if it is deliberately employed by an agent to help it achieve a goal which would otherwise be more difficult or impossible to accomplish.*

In most cases this idea coincides with Beck's definition, but it emphasises the fact that an object is a tool first and foremost because it helps an agent achieve a goal.

A *tool action* is an action which involves the manipulation of an object (the tool), with the effect of changing the properties of one or more other objects (the targets) or the agent itself, in a useful way. *By a 'useful way' we mean that the action has an effect which helps enable the preconditions of one or more other actions in the agent's existing library of actions.* The changed properties of the target objects which result from the tool action are called the sub-goals of the tool action.

A huge variety of tool-use actions fall within this definition. Examples include:

- nailing two pieces of wood together
- peeling a carrot with a peeler
- using a ladder to change a lightbulb
- using a torch to enhance perception
- using a brick to prop open a door
- writing with a pen
- using a cup to collect water
- catching a fish with a fishing line and bait

In this thesis we will restrict ourselves to tool actions which can be achieved with simple prehensile manipulation (ie. fixed to the manipulator), and focus our efforts on learning what the tool can help the agent achieve, what a suitable tool looks like, and the correct relative spatial pose in which it must be applied to be effective.

1.4 An example tool-use learning task

In this section we give an example of the type of problem we would like our tool using robot to be able to solve. This is followed in the next section by a description

of our approach to solving tool-use learning problems, using the same example as illustration.

1.4.1 The tube problem

In the tube problem the robot is set the goal of obtaining an object (in this case a small box) which is placed in a horizontal tube lying on the ground. The tube is open at one end and closed at the other, as shown in Figure 1.1. The agent is unable to reach directly into the tube to pick up the box because the tube is too narrow. In order to obtain the desired box it must use a hooked stick tool to first pull the box out of the tube before it can be picked up.

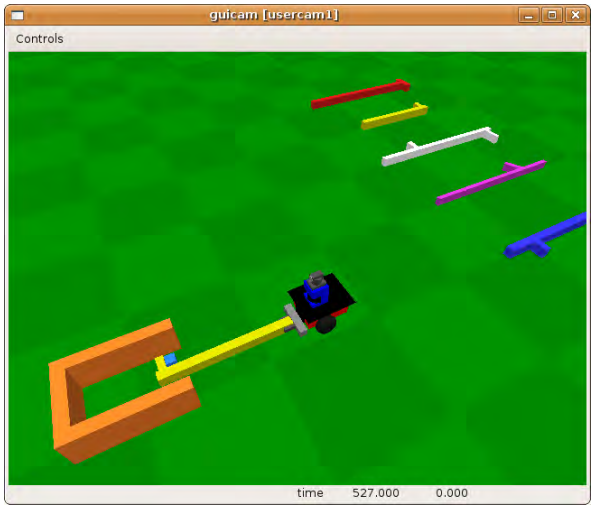


Figure 1.1: Using a pull-tool to reach an object placed in a closed "tube". This task is used in the experimental evaluation in Chapter 5.

The tube problem is interesting because it involves the use of a common type of tool used by humans and animals alike: a "reaching" tool, used to bring an out-of-reach object closer to the agent so it can be more easily accessed. In the animal kingdom, New Caledonian crows use hook tools to extract grubs from holes in logs, whilst chimpanzees extract termites by inserting thin sticks into passageways leading into termite mounds.

Variations on this task are also common in studies of animal tool use and cognition. Some of the earliest experiments with animals and tool use, performed

by Kohler and his chimpanzees (Kohler, 1925), involved using reaching tools to access bananas. Povinelli (2000) also placed a banana out of reach of a chimpanzee and provided it with different shaped stick and hook tools which it could use to retrieve its reward. Other researchers studying New Caledonian crows placed food rewards in horizontal glass tubes and provided the crows with a selection of sticks of varying length and thickness which could be used to either push (Chappell and Kacelnik, 2004) or pull (Chappell and Kacelnik, 2002) the reward from the tube.

As illustrated in the figure, we provide the agent with a selection of differently shaped objects which can potentially be used as a tools for solving the task. Some of these objects are clearly inappropriate, since they cannot be inserted into the tube, lack a suitable "hook" affordance, or are not long enough. However, the agent does not know ahead of time which objects will make a good tool. In fact, the only background information provided to the agent about specific objects is a low-level model of their geometric construction (ie. the object dimensions and relative orientation of its surfaces).

The agent is also provided with a set of useful behaviours which it can use to affect significant changes in the world. In this example, the agent might be given `goto`, `pickup-object` and `drop-object` behaviours. It should be emphasised however, that we assume that the agent does not already possess any sort of `pull-with-tool` behaviour — learning such a behaviour and using it to obtain the box is, after all, the objective of the problem!

To complement the robot's set of known behaviours, we provide a set of abstract models which represent the agent's knowledge of how executing each of these behaviours affects the world. For the purposes of this research our action models are written as STRIPS-like operators (Fikes and Nilsson, 1971). These action models can be used by the robot's planner to string together sequences of behaviours (plans) which will allow the agent to achieve its goals. An agent already competent at solving the tube problem might construct the following plan to achieve the goal:

```
goto(stick-tool), pickup(stick-tool), goto(tube),
pull-with-tool(stick-tool, box), drop(stick-tool), pickup(box).
```

The difficulty for our agent is that not only does it lack the appropriate tool-using behaviour (`pull-with-tool`) to solve the tube problem, but it also lacks the abstract action model of this behaviour. This means that the agent is initially unable to form even an abstract plan of how to achieve the goal of obtaining the box. This gap in both the agent's low-level behavioural knowledge and its higher level procedural knowledge presents the agent with a daunting exploration problem.

In this thesis we simplify the exploration problem in a reasonable manner, by providing the agent with an observation trace of a "teacher" performing the same task. Learning to use tools through demonstration is the most common form of tool-learning in humans, and cultural transmission of tool-using behaviour has been widely observed in animals (eg. (Krützen et al., 2005)). In the case of the tube example, our teacher agent demonstrates by simply picking up an appropriate stick-hook tool and using it to pull the box from the tube. The agent must learn from this example so that it can focus its direct experimentation in the world and make the exploration problem more manageable.

In summary, the problem faced by the agent is three-fold. Firstly, the agent must learn (from the demonstration provided by the teacher) that a new `pull--with-tool` action is not only possible, but would produce the useful effect of removing the box from the tube. Introducing a new abstract action in this manner would allow the agent to construct a plan to achieve the goal of picking up the box. This first step of the problem might be described as the *conceptual* step: learning what subgoals a tool can be used to achieve, and constructing an abstract plan of action.

Secondly, once the agent has learnt what the tool can do and constructed an abstract plan of action, it must learn how to actually perform the new tool-use behaviour which achieves the desired effect. In the tube problem, the agent must learn to place the hook tool in the correct pose in order to pull the box from the tube. This requires learning the correct spatial relations which describe the relative poses of the tool, target object, and the environment. For example, key spatial relations to learn in the tube problem are that the hook must be placed *behind* and *touching* the box and the handle should be roughly *parallel* to the

tube.

Finally, we would like our agent to be able to solve similar versions of this task in the future without having to repeat the whole learning process. In general, the agent will not always have the same objects available to use as tools and will never face exactly the same problem twice. If the agent can learn why a particular object is useful for solving the task it can avoid having to return to trial and error in finding a suitable tool for the new task.

In the tube problem the necessary properties of the tool include having a right-angled hook at the end of the handle, and that the hook is on the correct side of the handle (if the box is sitting in the left side of the tube, the agent needs a left-sided hook stick). By learning to identify these properties the agent should be able to select an appropriate tool from amongst a group of objects, and solve new tasks quickly and efficiently.

Learning the correct spatial and structural relations for tool use – the *generalisation* step – requires experimenting with a variety of tools and with variations in the tool-use task. Previous work on tool-use learning (which we will discuss in Chapter 2) has involved little or no generalisation across tools and tasks. In the next section we present our approach to tool-use learning, which *can* learn the necessary spatial and structural relationships between the tool components, target object and the environment.

1.5 An outline of our approach

In this section we give a brief outline of our approach to the tool-use learning problem, using the stick and tube problem described in Section 1.4 as an illustrative example. Full details of the learning methods are given in Chapter 4.

We divide our approach to learning into two components:

1. **Learning by explanation**: The agent tries to explain how the teacher was able to use the tool to solve the task. It uses this explanation to identify useful sub-goals of the tool action and construct an initial abstract action model.

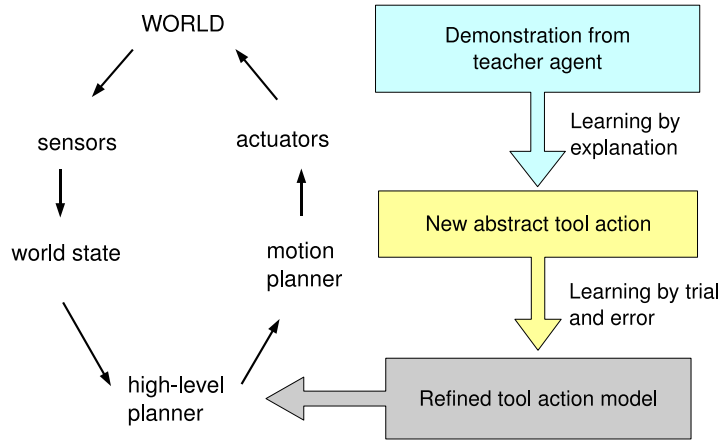


Figure 1.2: Illustration of learning in our agent architecture.

2. **Learning by trial-and-error**: Starting with the initial action model as its tool use hypothesis, the agent tries to refine the hypothesis by trial and error. The agent experiments with various tool objects and tries to learn the correct structural and spatial relationships which are necessary for achieving the tool action sub-goal identified in the learning-by-explanation step.

Figure 1.2 gives a simple illustration of how learning is incorporated into our agent architecture (the complete architecture is presented in Chapter 6). Learning is focused at the level of abstract actions which describe the structural and spatial constraints involved in executing a lower-level behaviour. Learnt abstract actions are translated into an executable behaviour via the use of a spatial constraint solver and a motion planner. We briefly describe this process before presenting our approach to learning from explanation and trial-and-error. Further details are given in Chapter 3.

1.6 Action representation

The agent's abstract action models are represented by four components:

- The objects involved in the action.
- A set of robot motion primitives.
- The preconditions of the action (expressed in first-order logic (FOL)).

- The effects of the action (also in FOL).

As illustrated in Figure 1.3, low-level behaviours are generated from an abstract action model via a constraint solver, motion (path) planner, and controller. This process involves extracting the abstract spatial goals of the action from the action model and passing them to a spatial constraint solver. The constraint solver takes as input a series of spatial goals expressed in first-order logic, and outputs a primitive state of the world which satisfies these goals. For example, if the abstract goal of the action is `on(book,table)` the constraint solver would output a precise position for the book on the table which satisfies this constraint. There are usually many primitive states which can be used to satisfy an abstract goal, and the constraint solver selects one of these randomly.

The world state which is output from the constraint solver is a specific primitive goal state which can be input into a motion planner to solve. The motion planner requires a set of motion primitives, which are supplied by the abstract action model, a list of the objects which are manipulated by the planner ("moving objs"), and a specification of their relative spatial pose during the manipulation. The planner then outputs a path for the objects which leads to the desired goal state. For example, for a robot pushing a box up against a wall the planner would output a path for the robot and box leading to the specific goal location. A generic controller is used to track this path.

The advantage of using a motion planner to generate behaviours (rather than learning purely reactive behaviours using, say, reinforcement learning) is that it allows the agent to quickly find solutions to novel subgoals. The downside is a loss of full generality and the ability to solve difficult manipulation problems involving fully closed-loop control. Since we are more interested in quickly finding "good-enough" solutions to simpler manipulation problems this is not a serious disadvantage. As stated in Section 1.3.1 we are more interested in learning how and why tools are useful at the abstract level. In addition our system architecture can be extended to incorporate more fine-grained closed-loop control learning, as described in Chapter 7.

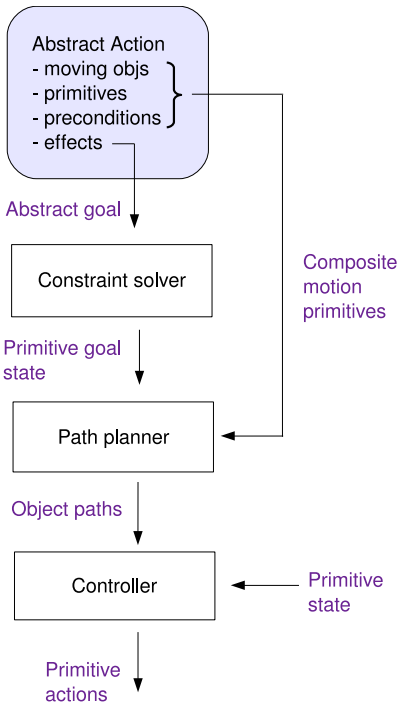


Figure 1.3: Illustration of how behaviours are generated in our agent architecture. The action model defines a set of composite object motion primitives which can be input into a path planner. The abstract goals of the action are extracted from the effects list and passed to a constraint solver, which produces a primitive goal state. The path planner finds a path to move objects to this goal state, and the generated behaviour is produced by a controller tracking the path.

1.6.1 Learning by explanation

The aim of the learning by explanation is to identify novel tool actions in the teacher's demonstration, and to construct an abstract action model which describes it. This includes idenitfying the useful sub-goal that the tool achieves, and some of the necessary preconditions for using the tool. It involves the following steps:

1. **Watch a teacher agent using a tool to complete a task.**
2. **Identify abstract actions in the teacher's demonstration.** The agent attempts to label the demonstration using the actions in its existing library of abstract actions.
3. **Introduce a novel tool action to represent unrecognised parts of the demonstration.** Any sections of the demonstration which cannot be

matched to known actions are labelled as components of a novel action.

4. **Construct a novel action model via explanation-based learning.** A STRIPS model for the novel action can be constructed by identifying the subset of literals in the novel action's start and end states which are relevant to explaining how the teacher achieved the goal.

Identifying novel tool actions in the teacher's demonstration

Our solution to the problem of recognising useful novel behaviours begins with the agent trying to explain what the teacher is doing during its demonstration. The agent constructs its explanation by trying to match its own set of abstract action models on to the execution trace of the teacher's demonstration. Gaps in the explanation – where the agent is unable to match an existing behaviour to what the teacher is doing – are designated as novel behaviours which the agent can try to learn.

This idea is illustrated in Figure 1.4 which shows a simple timeline representation of the teacher's demonstration for the tube problem. The parts of the timeline drawn in green correspond to periods of the demonstration where the agent recognises the teacher's actions. The "gap" corresponds to a sequence of one or more unknown actions which the agent must try and explain.

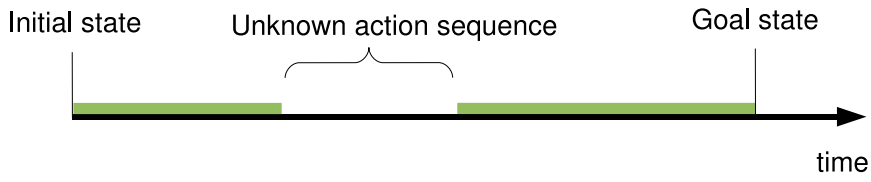


Figure 1.4: Illustration of an unrecognised action occurring during the teacher's demonstration. The green line represents the parts of the demonstration which were recognised by the learner.

The first difficulty faced by the learner is in labelling the parts of the demonstration which it recognises. The demonstration trace is provided to the agent as a discrete time series $w_1, w_2, \ldots, w_n$ of snapshots of the low-level world state taken every tenth of a second. This world state is comprised of the instantaneous

positions and orientations of each object in the world at a given point in time.

It should be emphasised that we do not provide the learner with a demonstration trace which is nicely segmented at the abstract level. Rather, the trace is a sampling of a continuous time series of object poses and the learner must arrive at a useful segmentation of the trace by itself.

As we explain in Chapter 4, the learner segments the trace into discrete actions by applying an object motion heuristic. This heuristic states that action boundaries occur at points at which objects start or stop moving. Thus when the agent grips a stick-tool and starts moving it towards the tube an abstract action boundary is recognised. When the tool contacts with the box and causes it to start moving also, another segment boundary is recognised. In this way the agent is to construct a very general movement-based description of the actions of the teacher. In the case of the tool problem the segments would be:

```
moving(robot), moving(robot,stick), moving(robot,stick,box),
moving(robot,stick), moving(robot), moving(robot,box).
```

These segments correspond to the robot moving to pick up the stick, placing it against the box, pulling the box from the tube, placing the stick down, and finally moving to pick up and carry away the box. Motion-based segmentation allows the agent to separate components of an unknown action sequence, whereas a purely STRIPS-based explanation would treat unknown sequences as a single "black box".

Once the learner has segmented the teacher's trace it attempts to match segments to its existing set of abstract action models. Each of the agent's abstract action models incorporates a STRIPS model along with a list of the objects which are moved during the action. A model is matched to a given segment by checking that three conditions are met:

- The moving objects in the demonstration match the model.
- The preconditions of the model are true at the start of the action segment.
- The effects of the model have been achieved by the end of the action segment.

The possibility of multiple matches to a single motion segment is discussed in more detail in Chapter 4. Segments which cannot be matched to any existing action model are labelled as components of a novel action. In the tube problem example this produces the following labelling of the teacher's demonstration:

```
goto(stick), pickup(stick), novel-action(stick,box),
drop(stick), goto(box), pickup(box).
```

where `novel-action(stick,box)` involves two unrecognised action components: in the first the stick is moved by the teacher, and in the second component the stick and box are moved together. The learner must now try and explain how this tool action was used to achieve the goal of acquiring the box. As we illustrate below, it does so by constructing a new abstract action model which is consistent with the other actions in its explanation of the demonstration.

Constructing the novel action model by explanation-based learning

We use an explanation-based heuristic to construct a model of the novel tool action observed in the teacher's demonstration. The explanation heuristic states that actions occurring before the novel action should enable the novel action preconditions. Similarly, the effects of the novel action should help enable the preconditions of actions occurring later in the demonstration. This heuristic is based upon the assumption that the teacher is acting rationally and optimally, so that each action in the sequence is executed in order to achieve a necessary sub-goal.

The program constructs a STRIPS model by examining the start and end states of the novel action in the demonstration, and identify relevant literals by examining the actions executed prior to and after the novel action. The effects of the novel action are defined as any state changes which occur during the action which support an action precondition later in the demonstration.

In the case of the tube problem, the effect $\neg$`in(box,tube)` occurs during the novel action segment as illustrated in Figure 1.5. This effect is considered useful or important because it enables the preconditions of the `pickup(box)` action which occurs later in the demonstration. In general there may be a large number of irrelevant effects which occur during an action, and the explanation-based reasoning

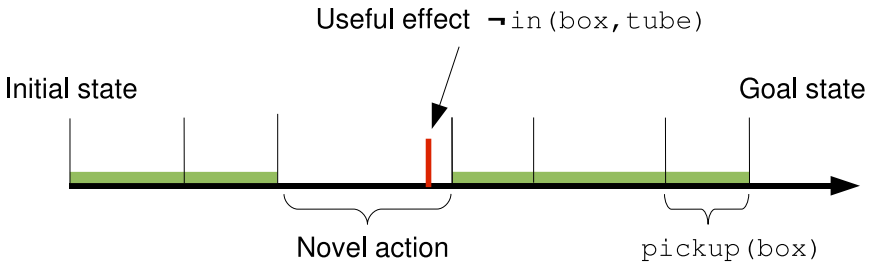


Figure 1.5: Illustration of a useful effect occurring within a novel tool action during the teacher's demonstration. We define a useful effect as one which enables the precondition of an action occurring later in the demonstration trace. In this case the effect ¬`in(box,tube)` enables the precondition of `pickup(box)`. We can use this explanation to identify the important effects of the novel action.

process allows us to identify the effects which were important to achieving the goal.

The preconditions of the novel tool action are constructed via a similar argument. The learner examines the world state at the start of the novel action, and identifies the subset of state literals which were produced by the effects of earlier actions. This situation is shown in Figure 1.6 for the tube problem. The effect `holding(stick)` occurs before the novel action, and is still true at the start of the action. Since it is a known effect of the `pickup(stick)` action it is considered a relevant literal to include in the novel action preconditions.

In the tube problem the novel STRIPS model constructed by explaining the teacher's demonstration is therefore as follows:

```
pull-from-tube(Tool,Object,Tube):
PRE:   in(Object,Tube), holding(Tool)
ADD:   -
DEL:   in(Object,Tube)
```

There is only one subgoal of the tool action in this case, corresponding to an object being removed from the tube. The ground preconditions and effects identified in the demonstration have been converted to literals with parameters, by simply replacing each instance of an object with a unique variable. It should be noted that the action model above is an incomplete model of the new tool use action,

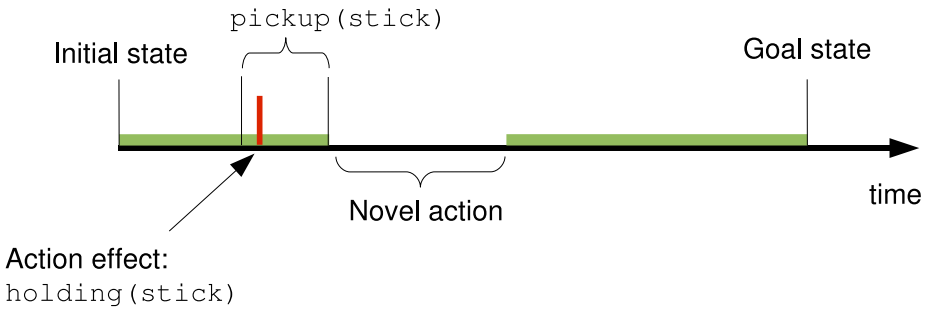


Figure 1.6: The preconditions of the novel action model are constructed by examining the effects of actions occurring earlier in the agent's explanation. Any effects which are still true when the precondition state of the tool action is reached are included in the action model preconditions. In this case `holding(stick)` occurs as the result of an earlier action, so it becomes a literal in the precondition.

since it says nothing about the spatial and structural constraints which must be satisfied by the tool. Nevertheless it is a good starting point for learning these additional conditions, which we describe in the next section.

1.6.2 Learning by experimentation

The novel tool action model learnt from explanation is just a starting point for the action learning process. As mentioned above, the initial model does not include all of the necessary spatial and structural information which describes the tool and the way in which it should be employed. The agent therefore attempts to refine this initial action model through learning by experimentation, testing a variety of tools and tool poses in order to solve a series of new learning tasks. In this section we describe this trial and error learning process, the details of which are presented in Chapter 4.

We define the state in which the tool is applied to the target object (which we call the tool pose state) as a concept learning problem — where the concept examples are generated and labelled through experimentation. A positive example of the concept is generated when the agent places the correct type of tool in an appropriate spatial position and manages to pull the box from the tube (achieving the action sub-goal). Conversely, negative examples arise from failures to achieve

the action sub-goal, which may occur due to the wrong tool being selected or the tool being placed in the wrong pose. Figure 1.7 shows three examples generated in the tube problem. It is important to note that the agent labels its own examples, rather than having them labelled by the user. The labelling is done by simply observing whether the tool action subgoal (identified in the learning by explanation step) was achieved.

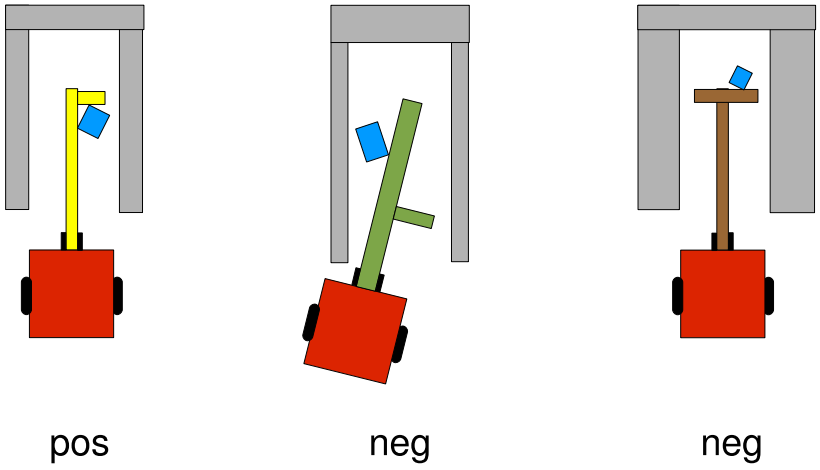


Figure 1.7: Positive and negative examples of correct tool use with a pull-tool. The leftmost example shows a positive example, whilst the other two are negative. Note that a correct example requires satisfying both structural constraints (which describe how the tool is constructed) and spatial constraints (which describe how the tool should be placed). If either is incorrect then the tool sub-goal will not be achieved.

The process for learning the concept describing the tool pose state is as follows:

1. **Test the current hypothesis:** Select a tool which satisfies the hypothesis and place it in a pose defined by the spatial constraints in hypothesis. Generate a motion plan which solves the task from this state and execute it, observing whether the action sub-goal is achieved.
2. **Add a new positive or negative example:** If the action achieved the desired sub-goal, label the initial state as a positive example. If the action failed to achieve the desired sub-goal, label it as a negative example.
3. **Update the hypothesis:** Run a relational concept learner to update the

definition of correct tool pose state.

4. **Generate a new learning task, or reset the current one:** If the current task was successfully solved, a new learning task is generated and presented to the agent. If the agent failed then the current task is reset.
5. **Repeat:** The agent continues its experimentation on a sequence of learning tasks, refining its hypothesis. The experiment is terminated when the agent is able to solve a pre-defined number of consecutive tasks without failure.

For each learning task presented to the agent, a selection of tool objects are available for solving the problem. The dimensions of the tools are constructed randomly according to a type definition specified by the user. Figure 1.8 shows the type of tools which are available for the tube task. Only the two hook tools (on the left of the figure), with right-angled hooks at the end of the handle are suitable for solving the task. Exactly which should be chosen in any particular case (a right-sided hook or a left-sided hook) depends on which side of the tube the box is located (right or left). Our relational learner is able to learn this important distinction, which would defeat a propositional learner.

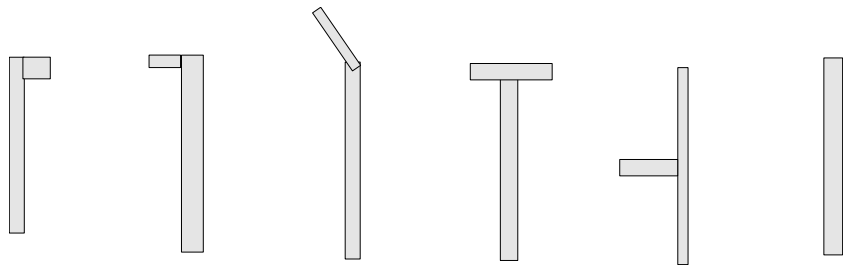


Figure 1.8: Examples of tools available for the pull-from-tube task. Only the two hook sticks on the left of the figure are suitable tools, and the exact one which should be chosen depends on the location of the box in the tube. A box on the left side of the tube requires a left-sided hook and vice-versa.

The application of relational concept-learning to learn a novel action is not new (see Chapter 2 for a discussion of previous work). However, the learning algorithm used in our agent is a novel one, based on previous work in the Inductive Logic Programming system GOLEM (Muggleton and Feng, 1992). We generalise over

examples in a bottom-up fashion using a form of restricted least-general generalisation (lgg) (Plotkin, 1970).

Hypotheses describing the correct tool and tool pose are represented as clauses in first-order logic. For example a hypothesis saying that the tool must have a hook which is touching the box would be:

$$h \leftarrow \texttt{attached(Tool, Hook),}$$
$$\texttt{touching(Hook, Box).}$$

Our contribution is to use a version-space-like representation of the hypothesis, where we maintain both a most-specific and most-general clause representing the concept. Our representation is not a true version-space as defined by Mitchell (1982), because we only keep a single clause to represent the most-general boundary (this point is discussed further in Chapter 4).

The motivation for using this hypothesis representation is that it is usually the case that the least general generalisation of the previously observed positive examples is too specific to be applied to the current situation — for example it may describe a tool with a specific combination of properties which does not exist in the current task. By maintaining both a most-general and most-specific clause we are able to search for a tool which avoids the previously encountered negative examples (satisfies the most-general clause), and is very similar to previously encountered positive examples (is "close" to satisfying the most-specific clause).

Our version-space-like hypothesis is illustrated in Figure 1.9. Since our learner must generate experiments to find positive examples, the aim is to generalise conservatively by trying to test examples which are close to the most-specific boundary (where positive examples are more likely to be found).

1.7 Experimental method

As detailed in Chapter 6 the platform we are using in this research is a Pioneer 2 robot (see Figure 1.10), and our experiments are carried out in the Gazebo robot simulator (Koenig and Howard, 2004). Although our robot platform is quite limited in its ability to manipulate objects, it nevertheless clearly demonstrates

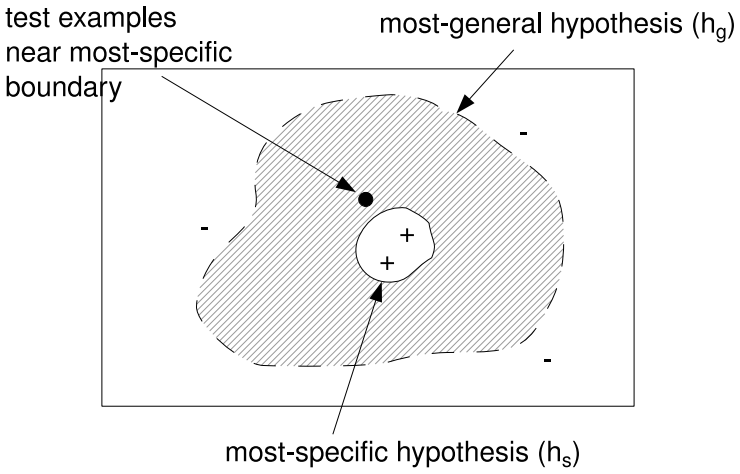


Figure 1.9: The most-specific (h_s) and most-general (h_g) boundaries on the hypothesis space. The + and − symbols in the figure represent positive and negative examples respectively.

the idea that tools can enhance the range of problems that a robot agent is able to solve. Indeed, agents with limited effectors often have a lot to gain by employing tools to overcome these limitations.

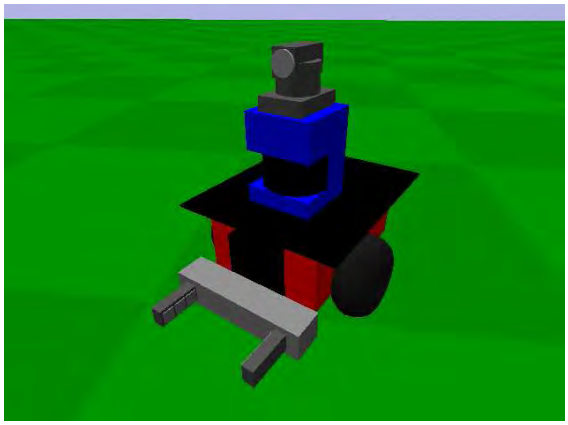


Figure 1.10: Pioneer 2 robot used in our experiments.

Our experiments consist of tool-use tasks such as the tube problem described earlier in section 1.4.1. The agent is presented with a demonstration of the task carried out by a teacher, and then is allowed to experiment on a series of new

tasks. We evaluate the agent's performance by the number of examples required before it can consistently solve any new task presented to it. Our experimental evaluation and a worked example is presented in Chapter 5.

1.8 Research objectives and scope

The primary research objective of this work is to develop a robot learning architecture capable of solving novel problems which require tool-use. The agent should be able to discover and exploit the useful properties of objects in order to achieve its goals.

The following design objectives are emphasised:

- The agent must be capable of learning from a single tool-use demonstration by a teacher.
- The agent should learn in an online and incremental manner, and from as few trial-and-error examples as possible.
- The agent should be able to autonomously identify the useful sub-goal(s) which can be achieved by using the tool (rather than having it externally defined by the user).
- The agent must learn to identify the properties of an object which make it a good tool, so that it eventually is able to select the correct tool without trial-and-error.
- The architecture and learning methods must be generally applicable to different robot platforms, and to a wide range of tool-use tasks.

It is also worthwhile pointing out some of the constraints and limitations we imposed on this research. Some important areas which this thesis does not address are complex manipulation, non rigid-body tools and objects, and analogical problem solving. We comment briefly on each of these points below.

- Complex manipulation:
  Some tool-use behaviours require complex non-linear control of the tool or target object. Learning these type of behaviours is an entire field of research

in itself, and we do not address it here. Instead, we have deliberately chosen tool-use tasks involving simple manipulation which can be achieved with a generic controller. Our learning is aimed at the higher level of identifying the structural and spatial relationships between objects which are required for successful tool-use. Nevertheless, in Chapter 7 we discuss how the learning of primitive behaviours could be incorporated in future work.

- Non-rigid body tools and objects:
  The learning tasks we tackle in this thesis are restricted to those which can be easily implemented in a rigid-body simulation (our experimental platform) — problems involving liquids or deformable bodies are not considered. Nevertheless, the general learning system described in this thesis is intended to be applicable to a wide range of tool-using scenarios. In Chapter 7 we discuss how our approach might be extended to cases involving non-rigid bodies, and illustrate it with some examples.
- Analogical problem solving:
  Some tool-use problems are related on an abstract conceptual level. For example, using a ladder to reach a light bulb and using a stick to reach an object on a shelf are conceptually similar problems. Once the agent has learnt to solve one of these tasks, can it solve the second by analogy? We do not attempt to tackle this problem in this thesis.

Although each of the above features of a tool-learning robot would indeed be useful, inevitably a research project such as this must be limited in scope.

1.9 Contributions

The primary contributions of this research are:

- The use of a relational learner for solving robot tool-use learning problems. Previous work in the area of robot tool-use learning has used a propositional representation and learning algorithm. As a consequence these learners are

unable to learn relational concepts which can distinguish between good and bad tools, and correct and incorrect usage of a tool.

- The integration of tool-use learning and problem solving in a robot agent. Our agent learns new tool-use actions in order to help it solve a planning problem and achieve its goal. Previous work on robot tool-use has been done in the context of learning isolated behaviours defined by the user.
- A novel action representation which integrates symbolic planning, constraint solving, and motion planning. The abstract actions in our agent are translated into executable primitive behaviours through the use of a constraint solver, which allows behaviours to be generated in a very flexible way.
- A novel method for incremental learning of concepts represented by Horn clauses. Our learner incorporates aspects of the GOLEM system (Muggleton and Feng, 1992) but uses a version-space type representation. We present a new method for generating examples in an incremental setting where cautious generalisation is desirable. Our approach is based upon sampling near the the most-specific hypothesis boundary in the version space.
- A novel method for learning STRIPS action models via (a form of) explanation-based learning. Our agent is able to infer the approximate form of a new action model by observing a teacher and examining the context of the plan in which it is executed. (Our approach also uses trial-and-error to refine this approximate model).

1.10 Overview of the thesis

The remainder of this thesis is organised as follows:

Chapter 2 presents a discussion of previous research relevant to the problem of tool use and learning in robotics. We discuss how this research differs from earlier work, and further motivate the approach we have chosen.

Chapter 3 describes how states and actions are represented in our tool-use

learner, including a novel approach to generating executable behaviours which satisfy a given abstract action model.

Chapter 4 describes the learning algorithms we have developed for learning new tool-use actions. The first half of the chapter presents our approach to learning from the demonstration provided by the teacher agent. The second part of the chapter describes how our agent learns relational tool concepts via trial-and-error.

Chapter 5 gives an evaluation of the learning methods presented in the preceeding chapters. We describe the experimental results obtained from using our approach to solve tool-use tasks in a simulated robot world.

Chapter 6 presents some of the implementational details of our agent, and shows how the execution, planning and learning components of the agent are integrated. The robot platform and simulator used are also detailed in this chapter.

Finally, **Chapter 7** summarises the main lessons of this research and presents suggestions for future work.

Chapter 2

Background and related work

In this chapter we review previous work related to tool-use learning. We begin by examining research which directly addresses the problem of tool-use learning in robotics, and follow with a discussion of related work in the area of action learning.

2.1 Tool-use and object manipulation in robotics

2.1.1 Robot tool-use learning

Despite the wide range of research which has explored tool use in animals and humans (see eg. (Baber, 2003)), to date there has been only a modest amount of research on learning tool use in autonomous agents and robots. In this section we review previous research in this area. Related work on learning robot manipulation of objects is summarised in Section 2.1.2.

A useful way of categorising robot tool-use learning approaches is by the type of model used to represent the tool. In general we can identify three different types of model used in research to date:

- *geometric*: Geometric models involve an explicit (though perhaps approximate) representation of physical geometry of the tool.
- *functional*: Functional approaches describe tools by the types of action outcomes they can be used to achieve. For example, a book and a cup can both serve as a paper-weight tool even though they have quite unrelated

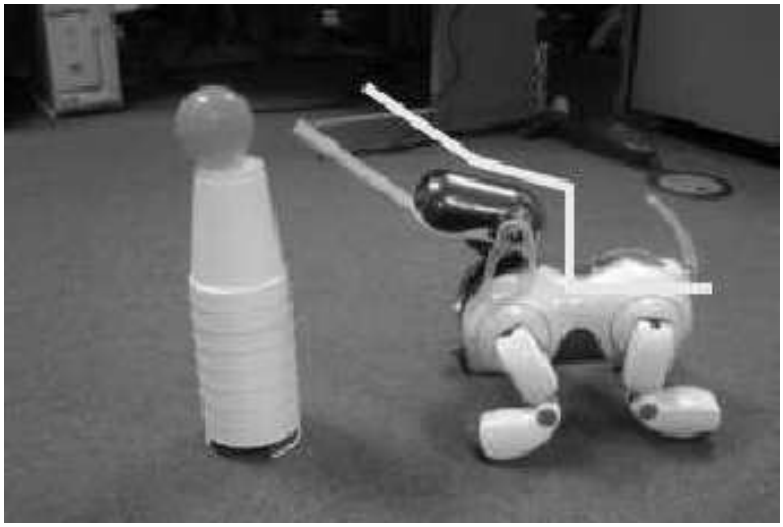


Figure 2.1: An AIBO robot using a stick tool to knock a ball off a platform in Wood (2005). The agent uses a geometric model consisting of four line segments representing the robot body, neck, head and the tool (as shown in the photo). The tool is considered to be an extension of the agent's body, and the same recurrent network is used to control the posture whether the tool is present or not.

geometric descriptions.

- *relational*: Relational models explicitly describe the structure and relationships between tool parts and/or the tool, target object and environment. A single relational model is able to describe a wide range of instances of a tool (eg. hammers) by explicitly encoding the important relationships which make the object a tool (eg. in the case of a hammer: a flat, heavy hitting surface attached to a handle at right angles).

Tool-learning approaches that use each of these tool representations are described below.

Geometric approaches

The first example of a robot learning with tools appears in Bogoni (1995). In this work the robot experiments with a series of differently shaped tools to learn their suitability for piercing and chopping tasks. The experiments involved simple manipulations in which the tool was lowered into contact with objects made

from different materials (styrofoam, sponge, pine and balsa wood). Observations from force sensors allowed the construction of a time-varying force profile of the interaction, while vision and position sensors were used to determine the success or otherwise of the chopping/piercing operation.

The visual data were also used to fit a simple geometric model (with two parameters) to the edge/tip of the tool and to thereby characterise the shape of the object. This information was then integrated into a multi-dimensional force-shape map for each tool-object combination where different points in the space represented variations on the object shape with their corresponding force and depth of penetration produced. By interpolating between the points sampled by experiment, a surface relating the shape, force, and depth of penetration of the material was constructed. The best shapes for the tool were selected by analysing repeated experiments and selecting shape parameters which led to the largest success rate for the task.

Another pioneering effort in building a tool-using robot agent is described in Wood (2005)[1]. In this work a Sony Aibo robot dog uses a stick-tool to dislodge a ball which is located on a platform out of direct reach, as shown in Figure 2.1. The agent uses a simple geometric model consisting of four articulated line segments to represent its body, neck, head and (optionally) a stick-tool. An interesting feature of this approach is that an MMC (Mean of Multiple Computations) network (Cruse, 2003) is used to encode the geometric relationships between the segments and to calculate the angles necessary to achieve a goal. This type of network, which is commonly used for the control of multi-segmented manipulators, is able to adapt to changes in the tool length segment without requiring recalculation of the network. Thus the same network can be used for calculating robot configurations whether the tool is present (ie. tool segment length of zero) or not. In this way the tool is treated as a natural extension of the agent's body.

Kemp and Edsinger (2006) have completed some interesting work on learning tool manipulation from a human demonstration, focusing on detection and control of the tip of an unknown tool. The tip of the tool is automatically extracted from

[1]Although the research in Wood (2005) does not involve robot learning, we mention it here because it involves an approach to flexible adaptation in a tool-using robot.

image data by using a multi-scale spatio-temporal interest point operator which selects fast-moving convex shapes in the image. A demonstration of the desired tool-using behaviour is then provided by a human, and the robot attempts to mimic the tracked trajectory by using a form of feed-forward control on the arm and wrist joints. Using this method the humanoid robot is able to learn how to clean a flexible hose with a brush without any prior models of the object or tool. This approach is an example of a crude (but effective) geometric approximation of a tool, where all of the relevant information is encoded in the location of the tool-tip relative to the robot arm.

Approaches to learning from demonstration which do not directly involve tool use are described in Section 2.2.4.

Functional approaches

A primarily functional approach to learning to use tools has been adopted by Stoytchev (2005, 2007). The robot arm apparatus used in this research is shown in Figure 2.2. The robot investigates the effects of performing user-defined primitive behaviours (pushing, pulling, or sideways arm movements) whilst applying different stick tools to an orange puck. The displacements of the puck which result from different combinations of tools and behaviours are stored in a look-up table, and this table can be used to solve simple manipulation tasks (such as moving the puck into the brown goal zone on the table shown in the figure).

Two interesting features of Stoytchev's approach should be noted. Firstly, by associating probabilities with the success of particular tool/behaviour combinations the agent was able to adapt when the tool was modified. That is, when one of the branches of a T-shaped stick was broken (removed) the agent was able to learn that the tool affordance it was using was no longer reliable. It was then able to adapt its tool behaviour by choosing an alternative affordance so that it could still move the puck into the goal location.

The second interesting feature of Stoytchev's work is that the robot does not possess a geometric model of the tool — it has no idea of the shape or structure of each object, and simply labels each by its colour. The characteristics of a tool are then encoded purely in the action outcomes which can be achieved by using

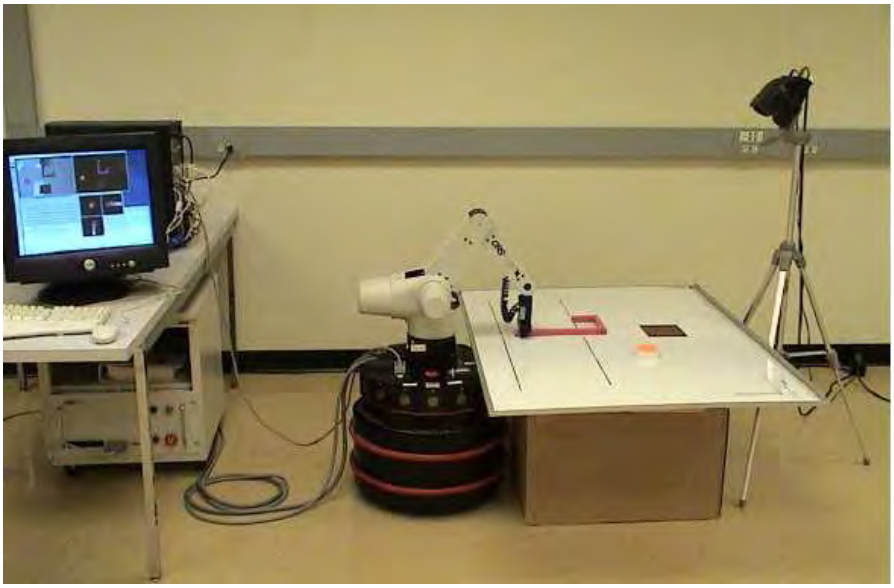


Figure 2.2: Experimental apparatus used in Stoytchev (2007). The robot arm is given a goal of pushing the orange puck on to the brown square using the tool. An interesting feature of Stoytchev's functional approach is that no geometric model of the tool is used to achieve this goal.

the tool. This is the essence of the functional approach to tool-use.

Sinapov and Stoytchev (2008) extended the approach of Stoytchev (2007) to learn functional taxonomies of tools. A simulated robot arm manipulated a puck using a variety of differently shaped stick tools and observed the different ways in which the puck moved. A hierarchical clustering algorithm was then used to find clusters of similar displacements which could be summarised by a prototype displacement. The set of these puck displacement prototypes for each tool were used to calculate a similarity metric which allowed comparison of one tool to another. In this way the agent is able to classify different types of tools based upon the functional abilities they possess. A large number of trials (1200) were needed to generate the clusters of outcomes, and these experiments were conducted in simulation.

Relational approaches

The disadvantage of the geometric and functional approaches described above is that generalisation across different objects and situations is difficult. For example, in a purely functional approach such as Sinapov and Stoytchev (2008) a robot must experiment and learn with each object before it can determine what type of tool it is. Kemp and Edsinger (2006) allows for generalisation across different tools, but only by abstracting away almost the entire detail of the tool and reducing it to a point representing the tool-tip. Bogoni (1995) learnt the object shape parameters which gave the best performance on a tool-use task, but this propositional approach is only effective when the important features can be represented by a compact set of attribute-values.

Tools are structured objects where the relationship between the constituent parts – and between the user, tool and target object – is critical to the effective functioning of the tool. A hammer, for example, should have a flat head which lies at a right-angle to the handle; the handle should be held near the base, and the flat surface of the hammer should be brought into contact with the blunt end of the nail, which in turn is held at right-angles to the piece of wood. Learning this sort of information in propositional (attribute-value) form is difficult. The learner would need to represent every possible relationship between each part of every object by an attribute — leading to a combinatorial explosion of attributes with no structure in the search space. Furthermore it is difficult to incorporate background knowledge into a propositional learning approach, and many relational concepts such as "the object under the hammer" cannot be properly expressed.

Due to the strongly relational nature of the tool-use domain we argue that some sort of relational learning approach is desirable. To the best of our knowledge, the work in thesis (first outlined in Brown and Sammut (2007)) is the first research in robot tool-use learning which uses a relational approach. Related work on relational approaches to learning models of abstract actions is covered in detail in Section 2.2.2, whilst an approach to discovering the relational structure of simple articulated objects through manipulation is discussed in 2.1.2 below. We also note that the problem of learning relational models of objects has appeared in other contexts, such as machine vision (Palhang and Sowmya, 1997).

As described in Chapter 3, our agent uses a relational representation of tool object and actions. Specifically we describe tools and situations in a subset of the language of first-order logic. Learning tool definitions and usage, such as in the hammer and nail scenario described above, is performed using a relational concept learner. This allows us to generalise across different objects and situations in a much more flexible manner than would be possible with a propositional learner (such as an attribute-value based decision tree). When our agent encounters a new object it is able to determine whether it might be a useful tool by considering the relationships which exist between its constituent parts, and the properties of these parts. Likewise, the way in which the tool should be made to interact with novel objects (and the user), is neatly captured by the relational representation.

Another feature of our symbolic relational approach is that it can be naturally incorporated into planning and problem-solving. In previous tool-use approaches (described above) the agent learns new behaviours but does not have to learn how or when they should be used. In our research the agent learns to use a tool in order to achieve a goal which it is otherwise unable to achieve. When a new behaviour is learnt it expands the range of planning operators which is available for problem solving. Related work on learning new planning operators and learning from demonstration is covered in Sections 2.2.2 and 2.2.4 respectively.

2.1.2 Learning object manipulation

Mason et al. (1989) and Christiansen et al. (1990) describe an early attempt to learn and use manipulation models with state transitions which are explicitly non-determinisitic. The authors describe a learning procedure for a robot manipulation task involving moving objects to goal locations by tilting the tray on which they are placed.

Lynch (1993) and Yoshikawa and Kurisu (1991) attempt to learn the friction parameters of pushed objects through direct experiment by pushing an object and observing the resulting displacement. This information, along with a geometrical model and a mechanics analysis, then allows planning of pushing manipulations, identification of stable pushing directions, and recognition of objects based on their friction characteristics.

Salganicoff et al. (1993) describes a system for a robot which learns to manipulate objects into new locations on a plane by pushing through a single fixed point contact. The agent learns a manipulation model which maps tangential pushing velocities to the resulting rotations of the object relative to the pusher. In order to move an object to the target a nearest neighbour method is used to select the most appropriate action to correct deviations from the desired path caused by the inherent instability of the point-contact pushing system.

Salganicoff et al. (1996) uses a version of ID3 decision trees to learn to select useful approach directions for object grasping from visual information. The robot was supplied with superquadric shape approximations of objects and was able to rapidly learn appropriate approach directions which allowed the gripper to pick up novel objects.

Fuentes and Nelson (1998) presents a method for learning dextrous manipulation of objects using multifingered robot hands. An evolution / genetic algorithm-based strategy is used to learn a set of manipulation primitives, which correspond to translating or rotating a given object in a set of orthogonal directions. More general types of manipulation are then obtained by scaling and adding or subtracting these primitives and relying on the mechanical compliance of the robot hands to produce the desired effect.

Other work focussed on learning models of pushing actions includes (Zrimec, 1990) and (Narasimhan, 1995). Narasimhan builds a local controller by sampling the action space to create a forward map of actions to differential displacements, and then uses table-lookup to choose the optimal action corresponding to a desired control correction. This is similar to the approach adopted in Salganicoff et al. (1993).

Fitzpatrick et al. (2003) uses simple poking and prodding actions to learn the resultant effects on novel objects of unknown shape and properties. The robot learnt a mapping of the initial hand position to the direction of movement of the target object when it was pushed, pulled or poked. These position-direction maps were then used to deduce the required actions for simple goal-oriented behaviour such as pushing the target closer to another object.

Katz et al. (2008) presents a relational approach to learning the kinematic

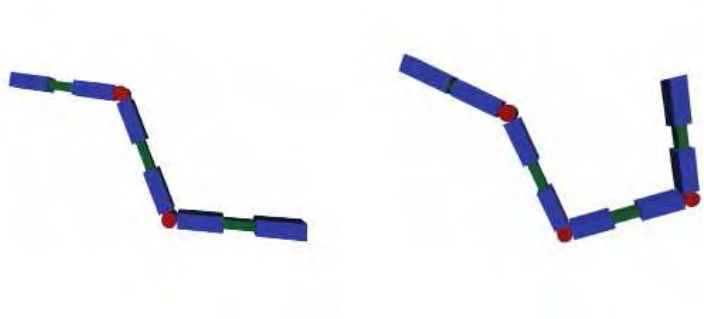


Figure 2.3: Katz et al. (2008) learnt simple relational kinematic models of articulated objects. The object on the left consists of two rotational (R) joints (in red) and three sliding (S) joints (in green). The object on the right possesses an extra rotational and slider joint. The agent is able to learn the kinematic structure SRSRS of the first object, and transfer this knowledge to learn the SRSRSRS structure of the second object more quickly than an inexperienced learner.

structure of articulated objects through manipulation. This is related to our research by virtue of its use of a relational approach, and the fact that many tools can be described as articulated objects (scissors, for example). Katz represents articulated objects by a set of links which are connected by rotational (R) or slider (S) joints. An example of an articulated object used in his experiments is shown in Figure 2.3. The leftmost object in the figure contains three slider joints and two rotational joints and can be summarised by the joint sequence SRSRS.

Katz's agent is given the task of learning the kinematic structure (the number, type, and arrangement of joints) in unknown articulated objects which it is given. Interestingly, the structure-learning problem is recast as relational reinforcement learning problem, where a reward is given to the robot each time it discovers a new link or joint in the object. Actions consist of choosing a link to push, and then observing the effects to see if any new links or joints can be incorporated into the agent's model of the object. By learning a policy which discovers structure quickly on one object the agent is able to use the same policy to quickly discover structure on related objects.

2.2 Learning actions for tool-use

Whilst there has been a relatively modest amount of research on learning new tool-based actions, there is a vast amount of related work on action learning in general. In this section we review these general approaches to learning actions, discuss their suitability in the tool-use domain, and explain where our approach to learning tool actions fits in.

2.2.1 Action-learning architectures

One useful way of characterising different approaches to action learning is to consider the level of abstraction at which learning takes place. Three-layer robot architectures (Gat et al., 1998) distinguish three levels of abstraction: the reactive layer, the mid-layer, and the deliberative layer.

Broadly speaking, the deliberative layer is the most abstract level and makes decisions by searching a model of the agent's environment — using a STRIPS planner for example (Fikes and Nilsson, 1971). In contrast, the reactive layer is the most primitive layer and involves no search. Its behaviours must respond quickly to changes in the agent's environment, and often involve direct state-action mapping or a feedback control mechanism. The mid-layer may use one or more of the reactive layer behaviours in sequence to achieve useful subgoals. Although the mid-layer may keep track of internal state variables in order to choose between reactive behaviours, it does no forward search. Both reactive and mid-layer behaviours can be modelled by the deliberative layer.

Our agent's three-level action architecture is shown in Figure 2.4. The deliberation layer consists of two sub-layers: the abstract planner FastForward (Hoffmann and Nebel, 2001) and a Rapidly-exploring Random Tree (RRT) motion planner (LaValle and Kuffner, 1999). The planner forms abstract plans in symbolic logic, which the motion planner then converts into ground paths through the state space. The mid-layer behaviours are responsible for tracking these paths, and they do so by repeatedly calling control-layer behaviours to achieve intermediate subgoals along the path. The control-layer behaviours, such as GotoPose, are implemented using a simple feedback control mechanism.

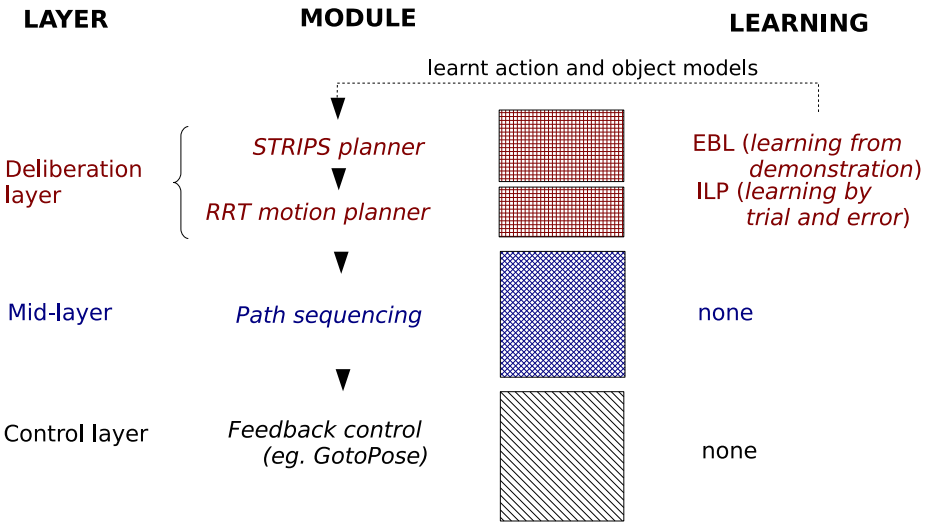


Figure 2.4: Our approach to a tool learning agent, represented as a three layer architecture. Note that learning occurs only at the deliberative level.

Learning in our action architecture occurs entirely at the abstract deliberation level, where abstract action models and qualitative models of tools are learnt. These models are learnt through a combination of learning by demonstration and learning through trial and error. The demonstration component uses an explanation-based learning (Ebl) type algorithm (Mitchell et al., 1986), whilst the trial and error component uses inductive logic programming (Ilp) algorithm to learn (Lavrac and Dzeroski, 1994). The object models allow the agent to identify objects which can be used as tools, whilst the action models describe the way in which the agent, tool and subject must interact in order to achieve the subgoals of the tool action. Operationalised instances of these models are used in the motion planning sub-layer and the results are fed down through the architecture to the mid and controller layers.

2.2.2 Learning abstract models of actions

As we have noted (see 2.2.1) our approach to tool-use learning focuses on relational learning at the abstract or deliberative level. In particular, our agent learns new tool-use actions in the form of abstract action models, which can be transformed

into low-level behaviours using a motion planner. Whilst there has been no previous work directly tackling the problem of learning relational models of tool-use actions, research on the more general problem of abstract action model learning is particularly relevant.

Approaches to learning abstract action models can be divided along a number of dimensions:

- the action representation and learning problem
- the environmental assumptions
- how examples are collected
- the learning method

In the remainder of this section we examine previous work on learning abstract action models, and STRIPS models in particular. The discussion is divided along the above dimensions, allowing us to highlight the differences and similarities between our approach and earlier methods.

Action representation and learning problem

One of the most common approaches to modelling abstract actions is the STRIPS representation (Fikes and Nilsson, 1971). STRIPS models, in one form or another, are the basis of most modern approaches to automated planning and as such STRIPS has been the favoured representation for learning abstract action models. Most of the learners cited in this section employ a STRIPS-like representation of actions (see Gil (1994) for example).

Under the STRIPS representation each action is represented by a STRIPS operator which consists of three main components:

- a list of *precondition* literals
- a list of *add* literals
- a list of *delete* literals

The precondition literals represent the conditions which must hold in order for the action to be executed. The add and delete literals together model the important

effects of the action. That is, when the action is executed the add list literals are expected to be added to the world state, while the delete list literals should no longer hold and are removed from the world state description.

Two types of learning task are commonly posed to a Strips model learner, differing on the amount of background knowledge supplied to the agent:

1. Starting from scratch, learn new Strips models of one or more actions (eg. Wang (1995); Benson (1996); Pasula et al. (2007)).
2. Given a partial or incorrect Strips model, learn the complete and correct model of the action (eg. Gil (1993)).

Approaches which learn from scratch, can be divided into those which attempt to learn action models with a single primary (intended) effect (Benson, 1996) or action models which take account of multiple possible effects (Pasula et al., 2007).

A less commonly posed learning task is to give the agent an initial action model and to ask it to learn a behaviour which fits the action model. This is an inversion of the usual problem of learning an action model of an existing behaviour. Ryan's Rachel system (Ryan, 2004) tackles this task by using a reinforcement learner to learn a behaviour which satisfies the precondition and goals of a Strips operator supplied by the user. A more conventional approach is then used to use learn side-effects of the learnt action.

Methods which learn abstract models of existing behaviours can be thought of as *bottom-up* learners, since the low-level behaviours must exist before the action model can be constructed. In contrast, approaches such as Ryan's are *top-down* in that the action model constrains the low-level behaviour to be learnt. Yik and Sammut (2007) is another example of a top-down learner — Strips models are used to apply qualitative constraints to a parameter space, and then a hill-climbing method is used to optimise within these constraints. In this particular case there is no further learning at the Strips level.

Whilst most action learners have focused on learning models of individual operators, other research has examined the problem of learning *sequences* of operators which correspond to useful tasks. Hume's Cap system (Hume, 1995) is one such learner which was used to learn procedures such as building an arch, juggling

balls, and climbing tasks involved in mountaineering. CAP represents procedures as concepts written in first order logic, where the construction and features of the concept describe how the procedure should be carried out. Buildling an arch, for example, involves first constructing a left column and a right column, and then placing a cross-beam on top of these columns. Other procedure learning systems are discussed in Section 2.2.4.

Benson's TRAIL system (Benson, 1996) learns both individual action models in a STRIPS-like representation, and arranges them in semi-universal plan trees to form useful reactive behaviours. The individual action models, which are referred to as teleo-operators, are a generalisation of STRIPS to continuous domains. Other interesting variations on the basic STRIPS representation include the work of Lorenzo and Otero (2000), where action models are learnt in the more general language of the situation calculus (McCarthy and Hayes, 1969).

The learning system presented in this thesis falls into the category of learning STRIPS action models from scratch. However, unlike other methods which learn new abstract models in a bottom-up manner, our approach can be thought of as a hybrid of top-down and bottom-up methods. That is, our system first learns the useful subgoal(s) of a new STRIPS action, before actually trying to implement a behaviour which achieves it. The initial action model is then revised by executing the action and observing the effects.

Environmental assumptions

The type of the environment in which the learner is designed to operate effectively is another key dimension of difference between the various approaches. The main questions which distinguish different learning environments are:

- are states continuous or discrete?
- is there noise in the world state description?
- is the world state fully observable?
- are actions stochastic?

Most systems (eg. Gil (1993); Shen (1994); Wang (1995); Lorenzo and Otero (2000); Pasula et al. (2007)) have no mechanism for learning from 'primitive' nu-

merical features — that is, numerical state predicates which can take on a continuous range of values, such as the precise spatial coordinates of a robot. Instead, these systems rely on the user to provide a set of discrete valued predicates (eg. `near(A,B)`) which discretise the world in a useful manner. Other systems, such as Benson's Trail, can learn from real-valued state literals in a limited manner. For example, in Benson (1996) Trail learns abstract action models from continous state predicates in a flight simulator scenario. Trail is also one of the few action model learners which takes explicit account of the duration of actions. Most other systems treat actions as atomic units, in the same way they are used in planning.

The ability to learn from noisy state descriptions is particularly important in the context of learning action models in robotics. Early systems such as Live (Shen, 1994), Expo (Gil, 1993, 1994) and Observer (Wang, 1995) were unable to handle noise. These same systems also made the assumption that actions are deterministic. Later systems used more sophisticated learning algorithms (see below) which could learn effectively in noisy environments with stochastic actions. Benson's Trail was the first of these, but action stochasticity was handled in a limited manner.

Work by Oates and Cohen (1996) and Pasula et al. (2007) explicitly modelled stochasticity by allowing for Strips action models with probabilistic effects. Pasula's system was able to learn a compact set of stochastic action models to describe the dynamics of a block stacking task in a three-dimensional physics simulator (Pasula et al., 2004). In order to take advantage of the stochastic outcome information encoded in the learnt models a probablistic planner is used in place of a more conventional Strips planner.

Very few systems have tackled the problem of learning in partially observable environments. In almost every case the assumption has been made that the relevant aspects of the world state are available to the learning algorithm. Amir (2005) is the only work we have found in the literature which is able to learn action models in partially observable domains. However this approach is limited to the case where actions are deterministic.

Our learning system is able to learn in the presence of noise, but we do not explicitly model the stochasticity of actions. Instead we simply accept that some of

the agent's actions may be unreliable and occasionally fail, and allow our planner to rebuild the plan when necessary. One consequence of this assumption is that our system will not try to avoid states which could accidentally lead to an unrecoverable failure. In our experimental domains this has yet to become an issue. Our system also relies on the common assumptions that the world state is fully observable, and the state description is discrete.

Collecting examples

The manner in which learning examples are collected is a critical factor to consider when building an action learner. In the action learning problem there are essentially two ways in which the agent can acquire learning examples[2]: by *demonstration* or by *trial-and-error* exploration. In the demonstration case, a teacher gives the learner useful examples showing how the action can be executed. In Benson's TRAIL system the teacher gives the learner a new demonstration each time the learner is unable to generate a plan by itself. Other systems, such as Wang's OBSERVER, Hume's CAP, and the approach described in this thesis rely on a teacher to give the agent an initial demonstration, and thereafter the agent is left to learn by experimentation. Learning by demonstration is discussed in more detail in Section 2.2.4 below.

In the trial-and-error learning case the agent is responsible for its own exploration. Gil's EXPO – which begins with partial action models supplied by the user – relies entirely on generating its own examples. It tries to do this in an efficient manner by seeking to conduct experiments which will result in the maximum information gain possible. CAP is another system which places a heavy emphasis on agent-driven exploration.

A related question is how the learning process interacts with the collection of examples. In general there are two approaches: *incremental* and *batch*. In the incremental approach, the action model(s) being learnt are revised each time a new example is collected. This approach is interactive in nature, and allows the learner to change its behaviour or plan each time it acquires a new piece of information.

[2]In fact we can identify a third method, learning by receiving advice. However advice-taking systems usually use this advice as a learning constraint rather than an action model in itself.

Examples of incremental (or 'online') action model learners include Live, Expo, Trail, Cap and Observer.

The constrasting approach (batch) involves collecting all of the examples 'offline' — that is, before any learning is done. As a consequence there is no opportunity for exploration or collecting examples on the basis of the current hypothesis. Batch learners are usually aimed at problems where a large number of examples are needed in order to form a good approximation to the action model. The exploration policy which directs the manner in which experiments are conducted is fixed, and is defined by the user before example collection begins. Pasula et al. (2007) uses the batch method with on the order of a thousand examples in order to learn stochastic action models. In this case the large number of examples are required in order to properly distinguish between different probablistic action outcomes. Other batch learners include Lorenzo and Otero (2000); Amir (2005).

The disadvantage of the batch method in the robotics domain is that collecting examples is an expensive proposition (in terms of time, and wear and tear on the robot). It is desirable therefore for the robot to learn from as few examples as possible, in an incremental manner — which is the approach we take in our work. In fact, our learner is not a true incremental learner since it completely rebuilds the hypothesis each time a new example is acquired (a true incremental learner would modify the existing hypothesis rather than rebuilding it completely). However since it takes much less time to build a new hypothesis from scratch than to collect a new example this has not been a significant limitation in our research.

Learning method

The properties of the environment described above have led to four main varieties of learning methods in the literature, shown in the tree in Figure 2.5.

Almost all previous work on action model learning has been inductive in nature. In this approach, observations of the world state before and after the execution of an action are used as examples in a concept-learning problem. To learn the precondition of an action (with one or more known effects), for example, states from which the action succeeds are defined as positive examples, while states from which the action fails are labelled as negative examples. A variety of machine-learning

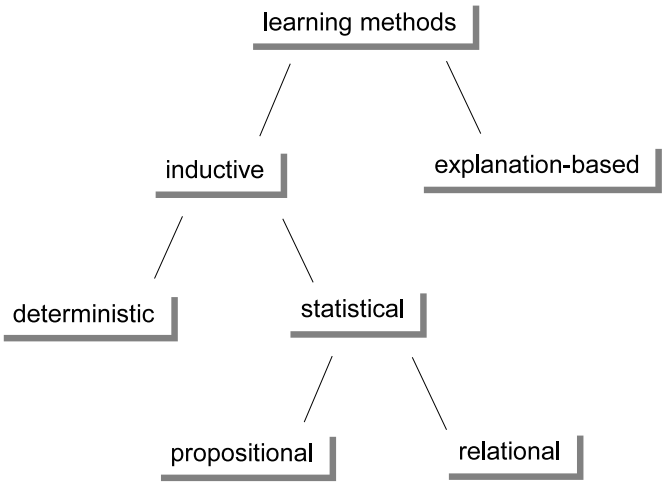


Figure 2.5: Categorisation of action model learning approaches. Our learning system uses both inductive and explantation-based algorithms. The inductive component is a statistical, relational learner.

algorithms can then be used to learn a definition of the precondition concept. The type of algorithm (deterministic vs statistical and propositional vs relational) determines the environments in which the learner can operate successfully and the concepts it is able to learn.

The early approaches to learning STRIPS models of actions were inductive and deterministic, using simple difference methods[3] to learn the preconditions of actions. Shen's LIVE system (Shen, 1994) learns the precondition concept by finding the difference between the current hypothesis and one or more misclassified examples. These differences suggest ways in which the concept can be amended. Gil's EXPO (Gil, 1994) uses a related method which compares negative examples to the most similar positive examples to find ways in which to modify the current hypothesis.

Wang's OBSERVER system (Wang, 1995) uses a variation of version-space based concept learning (Mitchell, 1978), where the algorithm iteratively amends the most-specific and most-general versions of the concept based upon the observed examples. More recent work by Amir (2005) uses a more complex and powerful approach to version-space filtering aimed at learning action models in partially-

[3]We borrow the term 'difference methods' from (Benson, 1996).

observable environments.

The primary limitation of these difference-based approaches was an assumption that the examples are noise-free and actions are deterministic. These learning algorithms have no mechanism for learning effectively in the presence of mis-labelled examples, as often occurs in a robotics context. Naturally a number of systems tackle this very problem using statistical concept learning methods developed in the machine learning community.

Oates and Cohen (1996) and Schmill et al. (2000) use statistical concept learning and clustering algorithms to learn the preconditions and effects of actions respectively. In Schmill et al. (2000), for example, a decision tree algorithm is used to learn the preconditions of actions. These algorithms were able to learn from primitive sensor data, such as sonar readings, where noise is a significant problem. The main restriction of this approach is that it is only useful for learning *propositional* Strips models — that is, models where the preconditions and effects are simply a list of logical propositions which are either true or false. The more general form of Strips model expressed in first order (relational) logic cannot be learnt.

In order to learn first-order models of actions in noisy environments, a number of systems have used algorithms borrowed from Inductive Logic Programming (Ilp) (Lavrac and Dzeroski, 1994). Ilp is a subfield of machine learning which is aimed at learning concepts expressed in a subset of the language of first-order logic. The general form of the Ilp problem involves learning a hypothesis H from a set of examples E, given background knowledge B, and under the assumption that H, E, and B are all written in first-order logic. Ilp methods can be used to learn action models by treating the world states as examples in a concept learning problem, in a similar manner to other inductive approaches.

Benson (1996) was the first to apply a noise tolerant Ilp algorithm to learning action models. His Trail system was able to learn the preconditions of actions in a variety of simulated domains in which noise would have defeated the difference methods used by earlier deterministic learners such as Gil (1994) — and unlike the statistical approaches in Oates and Cohen (1996) and Schmill et al. (2000) Trail could learn Strips-like models which were relational rather than propositional.

A number of other action learners have since employed ILP to learn models of actions. Ryan (2004) and Lorenzo and Otero (2000) both used variations of PROGOL (Muggleton, 1995) to learn action models in simulated worlds. In Ryan's work the ILP algorithm was used to learn the side-effects of STRIPS, whilst Lorenzo used it for learning action models in the situation calculus. Pasula et al. (2007) developed an algorithm which learnt stochastic relational rules in three-dimensional physics simulation. One of the distinugishing features of their approach was that the learning algorithm used a different search heuristic to the greedy hill-climbing methods used in previous methods.

Whilst most previous approachs to action model learning have used the inductive learning paradigm, an alternative approach is possible via explanation-based learning (EBL) (Mitchell et al., 1986). EBL has been used extensively in the STRIPS planning community for speed-up learning. In speed-up learning the aim is to learn rules for improving search speed or quality. This can be achieved by learning 'macros' for common sequences of actions, learning when to prune search branches, or learning better path selection heuristics. Of these only learning macros can be considered to be action model learning, and since these models only summarise sequences of known action models the learning is limited in scope.

Recent work by Levine and DeJong (2006) involves a more interesting version of action model learning based on EBL. In their approach, the agent is given simple analytical and qualitative constraints which express the relationships between domain variables. The explanation-based module of the learner uses the observed examples of system behaviour to convert this general background knowledge to a more compact relationship graph model. These relationship graphs represent the simplest causal explanation of the world dynamics. The graphs are then calibrated numerically and used to generate operators which can be used in planning. Levine's method is an example of operationalisation of a complex domain — learning a more compact and useful model of the world. It is however limited to learning in deterministic domains.

As noted by Kambhampati and Yoon (2008) it is popular misconception that EBL is not able to learn knowledge that is not already implied by the agent's background knowledge — and thus only speed-up learning or operationalisation of

a more general domain theory is possible. However, in the case where the agent's background knowledge is incomplete it is indeed possible to use explanations to guide learning of knowledge which is not already implied. This is the basis of the approach used in our work, where we use incomplete explanations of the teacher's demonstration to imply the relevant features which should be present in novel action models. These features could be learnt via a purely inductive process, but using an Ebl approach allows the agent to learn effectively from a single example — in contrast, an inductive method would require many examples in order to identify these features based upon statistical observation.

Where our approach fits in to action model learning

The work in this thesis describes a system which uses a combination of Ilp and Ebl algorithms to learn relational Strips of actions in an incremental manner. The closest existing action model learner to our approach is Wang's Observer system (Wang, 1995), which also learns incrementally and uses a version-space approach to representing the action model hypothesis. Wang's approach is nevertheless quite different to ours. Firstly, Observer uses a conventional version space representation which cannot handle any noise in the examples. In contrast, our approach is based upon a more robust Ilp algorithm which can learn in the presence of noise. Secondly, Observer operates in a purely discrete world and requires that the trainer provide an example trace which specifies the exact actions which were executed in each discrete state. Our system, meanwhile, takes as input a primitive state trace sampled from a continuous world and the trainer's actions are not provided to the learner. Thirdly, because our system incorporates Ebl it is able to learn from a single example, whereas in Observer more examples are required — in Wang (1995), for example, the learner is given 70 examples from the trainer.

2.2.3 Motion planning

Our system uses a motion planner to generate 'behaviours' for picking up and manipulating tools. The primary attraction of motion planning (compared with

behaviour learning) is that for a certain class of problems[4] it allows the agent to find solution trajectories to novel sub-goals quickly and effectively. For example, in a robotic arm reaching task the agent can generate a trajectory for grasping a novel object without requiring experimentation to generalise from previous experience.

There have recently been a number of papers exploring efficient combining task-level planning and motion planning (Cambon et al., 2009; Guitton and Farges, 2008; Stulp and Beetz, 2008; Stilman et al., 2007). These approaches seek to combine the advantages of knowledge-level planning for solving multi-step problems, with the flexibility of modelling at the geometric level offered by motion planning. The need for integrated task and motion planning systems arises particularly in the area of robot manipulation of objects, where the state space is too large to tackle with a pure motion planner and too complex to model exactly with a STRIPS level planner.

One interesting approach is the hybrid planner ASYMOV (Cambon et al., 2009, 2004) which combines a fast heuristic forward planner at the task-level with a roadmap-based manipulation planner at the geometric level. In this system simple STRIPS operators describe a relaxed version of the overall problem. That is, they carry no explicit geometric information and represent simple actions (pick up, put down, transfer object) which would be executable in the absence of any complicating geometric constraints. Geometric constraints, such as obstacles blocking the path, are resolved at the level of the manipulation planner. The symbolic planner is used primarily in the action selection mechanism, where the length of a solution plan gives a heuristic estimate of the cost of selecting that action.

The motion planner in ASYMOV uses a set of parallel roadmaps, one for each robot, movable object, or robot carrying an object. Links between roadmaps correspond to locations at which objects are picked up or deposited. Special *place* propositions in the STRIPS operators denote spatial regions or locations where roadmap nodes can be built — for example, P_CANGRASP_BOX might represent poses corresponding to locations from which the robot can pick up a box. The user

[4]In the context of tool use the motion planning approach is limited to prehensile tool manipulation — that is, tool manipulation where the tool object remains in a fixed pose relative to the agent's gripper. Non-prehensile behaviours such as pole-balancing can not be solved using a motion planner, and are usually tackled by nonlinear control or reinforcement learning methods.

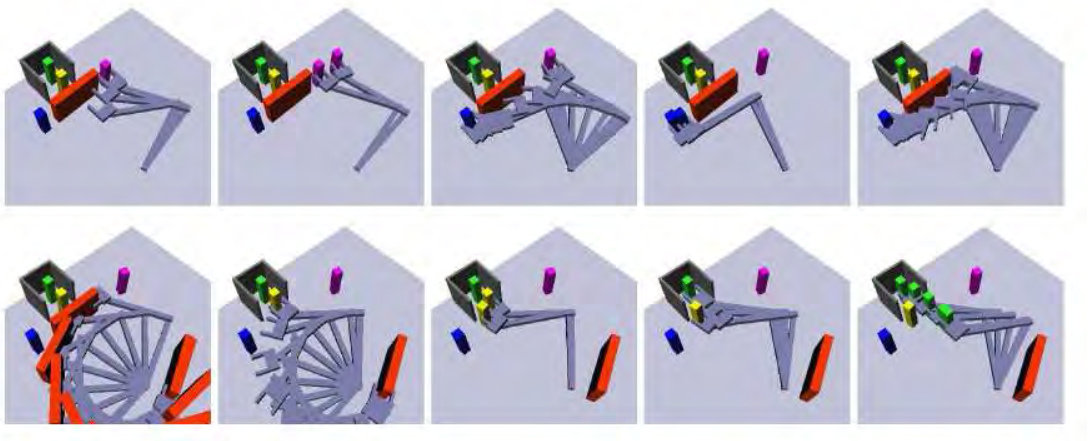


Figure 2.6: Sequence showing a solution for planning in the presence of movable objects, reproduced from Stilman et al. (2007). The goal was for the robot arm to obtain the green block.

defines a method for generating nodes for each *place* proposition, often involving simple inverse kinematics to generate the pose.

In contrast, in our approach the interesting regions of space (corresponding to the place propositions) are not all defined by the user in advance and must instead be learnt. In particular, the correct spatial affordances between the agent, tool, and objects are not known. As a consequence we cannot define simple methods for generating configurations which satisfy these relationships and must rely on a constraint solver to find valid poses for these roadmap subgoals. A further difference is that our "place propositions" are learnt as explicit relational expressions, rather than simple propositions which are implicitly defined by the user.

The recent work of Stilman et al. (2007) is also noteworthy in the context of this thesis. In this work the authors address the problem of planning to pickup and move an object which is surrounded by a number of surrounding obstacles. The obstacle objects must be moved aside in order to access the target object with the robot arm, and conventional obstacle-avoiding motion planners are unable to solve these sort of tasks. An example of a robot arm solving this problem is shown in Figure 2.6.

The approach used involves finding a path for moving the target object whilst

initially ignoring collisions with movable obstacles. This produces a set of objects which must first be moved out of the way in order that the target object can be moved without causing any collisions. A simple backward-chaining planner is used to consider plans which differ in the order in which the obstacles can be moved. Indirect obstacles (obstacles blocking obstacles) are also added to the set of objects to be moved. For each ordering alternative an efficient low-level motion planner is used to search for a solution which gives access to the target object.

The planner used in Stilman et al.'s work is task-specific and considerably less general than a STRIPS planner. Nevertheless, it illustrates the utility of employing a higher-level planner to guide low-level manipulation planning. Interestingly, we can consider the planner in Stilman et al. (2007) to be a specialised version of our planner, where the available actions consist of removing "obstructions" from the desired path of our agent. This would require defining (or learning) a geometric predicate describing the conditions under which an object can be considered a potential obstruction.

Guitton and Farges (2008) define three general approaches to implementing the interaction between a task planner and a motion planner:

- *hierarchical*: this is the naive approach, where a complete symbolic plan is computed, before being verified by a motion planner
- *hierarchical with backtrack*: similar to hierarchical, but when the motion planner fails the task planner backtracks to find a new solution (rather than starting again from scratch)
- *interleaved*: the task planner and motion planner exchange information after each step, so that information from one can be used to guide the other

Guitton and Farges demonstrate that the interleaved approach leads to significant speed gains over the naive approach. They implement a version of an interleaved system which exchanges messages with a common vocabulary between the task planner and motion planner. The messages consist of either constraints or advice which helps to guide the search. Our system implements a naive hierarchical approach to combining the task and motion planner. However, as described below,

our emphasis is on learning new Strips operators and manipulation primitives. It would be a relatively straightforward matter to extend our architecture to implement a more efficient planner integration as in Guitton and Farges (2008) or Cambon et al. (2009) in future work.

The approaches to integrating task-level planning and motion planning described above are focused on implementing an efficient interaction between the two levels. Our work addresses a different problem: learning useful new task-level operators and manipulation-level primitives. As far as we are aware ours is the only work in the literature which focuses on learning new Strips actions and motion primitives in an integrated system. Existing systems either use a fixed set of manipulation primitives and Strips actions (as above), or learn new Strips actions of fixed behaviours (as in Section 2.2.2). Unlike most other Strips model learners (eg. (Benson, 1996)) our system is also able to directly access the geometric model to test hypotheses during learning.

Our system can be thought of as a hybrid Strips and manipulation planning system which is able to learn new ways of interacting with the world. Each of our Strips models represents a subset of all possible motion plans, defined by symbolic constraints on the start and end states, as well as the manner in which objects are manipulated. Learning a new Strips operator is equivalent to learning a (hopefully) useful new way of moving objects around in the world. Although it is currently focused on non-prehensile object manipulation tasks we describe how the architecture can be extended to learn other types of behaviours in Chapter 7.

2.2.4 Learning from demonstration

Our agent relies on a tool-use demonstration in order to seed the learning process. Learning from demonstration is a broad field which has received a great deal of attention from the robotics community. Since we can only give it a relatively short discussion here, the reader is referred to Argall et al. (2009) for a more comprehensive survey of approaches to robot learning from demonstration.

Approaches to learning from demonstration can be divided along many dimensions — how the demonstration is recorded (directly vs indirectly), whether the teacher agent is identical to the learner, and what exactly is being learnt. Here we

examine approaches from the point of view of what is learnt. The most common task involves simply trying to replicate a primitive control behaviour in the form of a state-action mapping. Other learning opportunities include learning subgoals, constraints, reward functions, and procedures or activities. Previous work on robot learning from demonstration in each of these areas is discussed below.

Note that approaches to learning abstract actions from demonstration, which are most closely related to our work, were discussed in Section 2.2.2.

Behavioural cloning

Behavioural cloning (Michie, 1993) is a popular method for learning to mimic low-level control policies demonstrated by a (usually) human operator. In behavioural cloning the agent is given examples of desirable behaviour and must generalise these specific examples to new situations.

Two distinct approaches to behavioural cloning are possible: direct and indirect. In the direct approach to behavioural cloning no model of the system is built; instead, the agent attempts to learn a state-action policy by treating it as a concept learning problem. Sammut et al. (1992) describes an example of the direct approach where the agent learns to fly a plane in a flight simulator after observing the control traces of a human pilot. Each state-action pair from the control trace is considered to be a positive example of the desired behaviour, and the concept learner generates a classifier which tries to predict the correct action to take in any given state. In theory almost any machine learning algorithm capable of learning concepts can be used to learn a control policy in this manner — a decision tree learner is one popular approach, and is used in Sammut et al. (1992).

Indirect methods learn a partial model $x_{k+1} = f(x_k, u_k)$ of the system from the demonstrator's control trace, and then use that model to achieve the desired behaviour. Suc and Bratko (1997), for example, learns an approximate model of the system using locally weighted regression, identifies subgoals in the operators trace, and then constructs a series of linear quadratic controllers which achieve these subgoals. The method is used to learn to control a simulated crane. Potts (2007) describes a related approach where a linear model tree provides a piecewise-linear model of the environment and local controllers are used to follow a trajectory

to the desired goal.

Atkeson and Schaal (1997) uses both parametric and non-parametric modelling approaches to learn to control a pendulum. The parametric model is derived from the physics of an idealised pendulum, and parameters are learnt by regression of the observed demonstration data. The nonparametric model uses locally weighted regression to construct a series of local models from the training data.

From the point of view of our research, the main restriction in the conventional behavioural cloning approach is that an expert must provide a complete state-action control trace. In the case of flying a plane or driving a crane a suitable expert operator can easily be found and a control trace recorded. In the tool-use scenario however it is unreasonable to expect that an expert operator exists who can control the robot to achieve the desired goal. Rather, we want the agent to be able to learn by observing and imitating another agent (a human, for example). Under these conditions the agent observes the demonstration but does not know the exact actions which were executed by the teacher.

Learning subgoals or constraints

One of the objectives of our agent learning from demonstration is to learn the subgoals which can be achieved through using a tool. Subgoal based approaches have been used in behavioural cloning in order to overcome some of the limitations of the state-action mapping approach (Bratko et al., 1998). In particular, lack of robustness with respect to changes in initial conditions or variations in the environment. Subgoals try to capture intention of the expert demonstrator so that the agent is better able to respond to conditions which vary from that of the demonstration.

Isaac and Sammut (2003) apply subgoal learning to behavioural cloning of flying manuevers in a flight simulator with turbulence. Model-tree learners are used to learn both goal rules and control rules; the goal rules model the desired settings for their controllers and the control rule describes control settings which correct errors between the current state and the goal state. Segmentation of the demonstration into different behaviours is done manually, so the learner does not have to address the problem of knowing when the subgoal has changed.

In Morales and Sammut (2004) a demonstration trace is used to find a subset of predefined controller actions which might be useful for flying a plane. In this case possible subgoals were not identified per se, but rather a small set of relevant actions per state. A reinforcement learner was then used to learn a policy over this small set of useful actions and states.

Suc and Bratko (1997) define subgoals as the points in the state space at which the operator's policy changes. In the approach used in this research, these points were determined by finding minima in the weighted linear quadratic cost. A LQ controller was then computed to control the system for each stage between subgoals. The approach was used to control a container crane in simulation, a system with nonlinear dynamics.

Finally, Pollard and Hodgins (2002) take an alternative approach to generalizing demonstrated manipulation tasks by extracting constraints from the example. A human demonstration of manipulating (tumbling) a large box was used to define the number of manipulation contacts and constrain the forces exerted by those contacts. A specialised manipulation planner was used to find solutions to the task within these constraints.

Learning reward and cost functions

Another approach to learning from demonstration involves learning states or features that should be reached or avoided when attempting to mimic the task: that is, learning cost or reward functions. This area is often referred to as "inverse" reinforcement learning (Ng and Russell, 2000), since it involves finding the reward function given a demonstrated behaviour rather than the more conventional case of finding a suitable behaviour given a reward function.

Atkeson and Schaal (1997) presents a system which learns both a system model and a reward function after observing a demonstration. The problem involved a robot arm learning a pendulum swing-up task, where a pole is swung up and then balanced upright. The reward function and dynamics model was created from 30 seconds of human demonstration and punished deviations away from the demonstrated trajectory. The model and reward function were then used to learn a policy to achieve the task.

In Abbeel and Ng (2004) the agent learns a reward function under the assumption that it can be expressed as a linear combination of known features. The technique was applied to learning driving styles in a driving simulator. Abbeel et al. (2007) learns both transition model and reward function for performing aerobatic maneuvers on a small autonomous helicopter. A human pilot provides the demonstration so that the dynamics model and reward function can be built.

Ratliff et al. (2006) takes demonstration paths provided by an expert and attempts to learn a mapping of features to costs. An MDP or A* search algorithm can then be used to find paths in other instances of the same task. This approach is interesting because, like our approach, the agent is only given the sequence of states which the expert traverses — it does not know the actions which were executed. In Ratliff et al. (2006) the technique is used to learn to plan effective paths across terrain in satellite images.

Learning plans, procedures or activities

Learning abstract action models from demonstration was discussed in Section 2.2.2. In addition to this work there has been some robot learning from demonstration focused on learning at the higher level of plans, procedures and activities. The emphasis is on learning an abstract description of the teacher's actions which can be reused or generalised to other tasks. Two important questions are how the sequence of actions executed by the teacher is segmented into known sub-tasks or behaviours, and how the individual action segments are recognised.

In early work by Kuniyoshi et al. (1994) a robot torso learns reusable plans for assembly tasks – such as constructing a simple 'table' – by watching a human demonstration. The system uses qualitative events and features to segment the demonstration into a sequence of operations. Some changes in the initial state are possible (such as positions of objects) but the process requires that the final state and the parts used during the assembly are the same as the demonstration.

Veeraraghavan and Veloso (2008) present a method for teaching sequential tasks with repetition through demonstration. The "demonstration" in this case consists of instructions from the teacher of which actions to execute; relevant objects are indicated with a laser pointer. The agent therefore does not have to

do any segmentation of the demonstration or recognition of behaviours. Action preconditions and effects are calculated using simple difference methods (the algorithm does not appear to be designed to handle noise at the abstract level). The interesting feature of their approach is that it can learn to carry out plans (procedures) with loops. The approach is demonstrated by having a robot learn to repetitively put objects in a box.

Hume's CAP system (Hume, 1995) used the demonstration of a teacher to learn procedures such as building an arch, juggling balls, and climbing tasks involved in mountaineering. In this work the agent has access to the actions executed by the teacher, so the segmentation problem does not exist. The approach has yet to be applied in a robotics scenario.

Nicolescu and Mataric (2001) learn generalised procedures in the form of behaviour networks from multiple demonstrations of a task. The agent watches the demonstration and records the time periods when the postconditions of any of its behaviours are true during the demonstration. It then analyses the temporal ordering of these periods to determine if one behaviour produces permanent or enabling preconditions for another behaviour, or if an ordering constraint between the two behaviours exists. These relationships are represented in a behaviour network which can be used by the agent to carry out the desired procedures. The technique is illustrated by teaching a Pioneer 2DX robot to visit object targets in a particular order, move objects from one location to another, and to slalom around obstacles. Whilst Nicolescu and Mataric (2001) involves learning from a single demonstration, the ideas are extended in Nicolescu and Mataric (2003) to generalise the behaviour networks after multiple demonstrations. Verbal cues are also used during the demonstration to help the robot to attend to relevant information and help create correct generalisations of the task.

Although it does not involve action learning from demonstration, we also note here the work of Sridhar et al. (2008). They present an approach to unsupervised clustering of object-based activities which is related to part of our approach to learning from demonstration: specifically, the technique used for segmenting the activity into epsiodes is similar to our approach. In their work the segmentation is based on changes to the topological relationship between bounding boxes

Figure 2.7: A sequence of episodes making up "pickup object" sub-activity. A new episode occurs at the point where a spatial connectedness predicate changes (from Sridhar et al. (2008)).

of objects: a new episode occurs in places where bounding boxes become either disconnected, surrounding, or touching. Figure 2.7 shows five episodes which result from a hand reaching in to pick up an object on a plate. Our approach uses a coarser segmentation based on overlapping, moving bounding boxes and only segments at points where objects stop or start moving.

2.2.5 Learning primitive behaviours by trial and error

The trial-and-error behaviour learning problem has been addressed in a very general way in the fields of dynamic programming and reinforcement learning (Sutton and Barto, 1998). In the dynamic programming approach the agent has access to a complete model of the environment, and is given an implicit goal for the behaviour in the form of a reward function. The agent must learn a behaviour (a 'policy' which maps states to actions) which maximises the expected cumulative reward it receives whilst executing the behaviour.

Reinforcement learning is similar to dynamic programming, but tackles the case where the agent is not given an *a priori* model of the environment. Two flavours of reinforcement learning can be distinguished: *direct* and *indirect*. The

direct (or model-free) approach does not build a model of the environment and instead attempts to directly learn the best action to take in any given state — the so-called optimal value function. In contrast, indirect reinforcement learning involves learning a state transition model and using this model to compute the optimal state-action policy.

From the point of view of tool learning, the primary advantage of these approaches is that they would allow the agent to learn to perform a wider range of behaviours than can be represented by our STRIPS-plus-motion-planner approach. The corresponding disadvantage is that they are generally either slow to learn, or require careful tailoring of the state or policy representation for each behaviour to be learnt. Our use of a motion planner with symbolically defined motion primitives allows us to generate solutions to novel problems quickly, and requires only a single example to seed the learning process. Furthermore, the relational nature of the STRIPS action models means that the learnt actions can adapt to changes in objects and situations.

Chapter 3

State and action representation

In this chapter we describe our approach to representing states and actions for tool-use problems. We describe how abstract action descriptions, expressed in first-order logic, are translated into executable primitive behaviours.

3.1 Overview of the architecture

Our agent uses a two-layer representation of states and actions as illustrated in Figure 3.1. At the primitive level, the agent receives state information from the environment and executes primitive actions in response to this information. At the abstract level, states are described in first-order logic and actions are represented with STRIPS-like operators. A planner is used to construct useful plans to achieve the agent's goals and an action model learner amends existing action models or learns entirely new ones.

The primary point of difference in this architecture, as we shall see in more detail below, is the way in which abstract action operators are operationalised into useful low-level behaviours via a constraint solver and motion planner. Each action model defines a set of sub-goal states for the motion planner, and the constraint solver is used to find ground solutions which satisfy these sub-goals in the current state. This allows our abstract actions to be treated as *generators* of low-level behaviours, rather than simply being *models* of existing behaviours.

Our approach to generating low-level behaviour – using abstract STRIPS-like

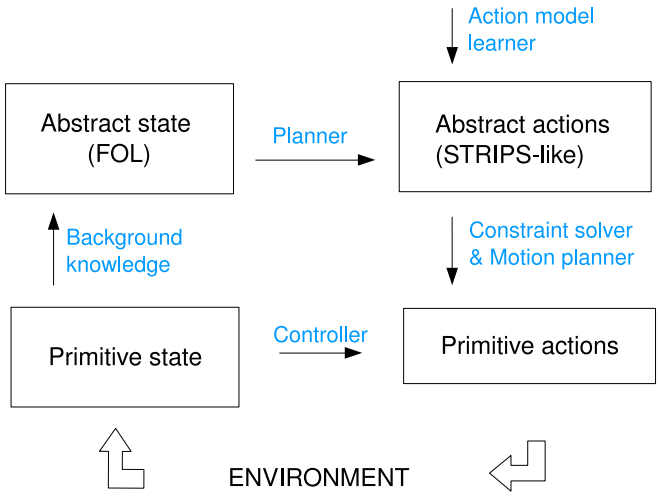


Figure 3.1: The agent uses a simple two-layer representation of states and actions. It receives primitive state information from the environment and replaces this primitive state description with an abstract state description based upon predicates defined in its background knowledge. High-level actions are represented by STRIPS-like models, and a planner is used to select the appropriate action to execute according to the current world state and the agent's goal. Abstract actions are translated into executable primitive behaviours through a more complicated process involving a constraint satisfaction solver and a motion planning algorithm. This process is described in detail in Section 3.3.2.

operators in conjunction with a constraint solver – is related (but quite different) to previous work by Yik and Sammut (2007). In Yik's work STRIPS models are used to constrain the search space for a low-level walking behaviour, although the STRIPS models are provided by the user rather than being learnt. In our work the action models are learnt, and operationalised via a constraint solver and motion planner.

3.2 State representation

3.2.1 Primitive state

At the low-level the primitive state consists of the positions and orientations of all objects in the world at each time step. In addition to this dynamic spatial information, the agent is given "static" information about the unchanging geom-

etry and structure of each object in the world. These data consists of the shape, dimensions, and attachment points for each object and its components.

3.2.2 Abstract (relational) state

On top of the primitive state the agent builds an abstract state description which is able to explicitly represent the complex properties and relationships which exist between objects and agents in the world. The abstract state description is expressed in first-order logic, and is generated from a set of predicate definitions provided to the agent as background knowledge. These abstract predicates, such as `on(X,Y)` are expressed in terms of primitive state variables (object poses) or other abstract predicates. The exact poses of individual objects are not represented explicitly in the abstract state, so a single abstract state description may be used to describe a (possibly large) number of primitive states — for example, an abstract state defined by `on(book,table)` describes many different primitive states in which the book is in different positions on the table.

An example abstract state predicate

As a simple example of an abstract state predicate we consider the spatial predicate `onaxis(Object,Robot)` which is true if `Object` lies on the primary axis of `Robot` as shown in Figure 3.2.

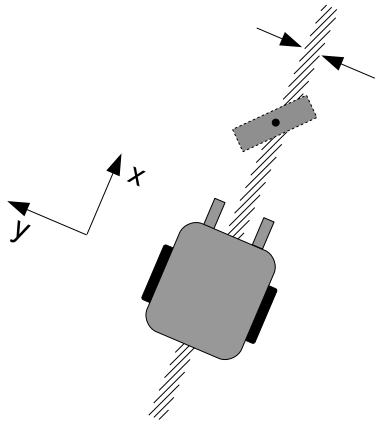


Figure 3.2: Illustration of the `onaxis` predicate.

The hashed area in the diagram shows locations at which an object will be on the robot axis. This predicate might be defined in terms of the primitive poses of the two bodies as:

```
onaxis( Object, Robot) :-
      pose( Robot, RobotPose),
      pose( Object, ObjectPose),
      relpose( RobotPose, ObjectPose, [X,Y,Th]),
      abs( Y) < 0.05.
```

In plain English this predicate states that if the pose of the object relative to the robot is `[X,Y,Th]` then the Y-displacement must be less than 0.05m. Here `relpose(Pose1,Pose2,[X,Y,Theta])` is a geometric primitive which calculates the relative pose `[X,Y,Theta]` between two poses, using the first pose as the origin of the coordinate system. Meanwhile `abs(Number)` simply gives the absolute value of an expression.

3.3 Action representation

As illustrated earlier in Figure 3.1 our architecture uses two layers of actions: abstract and primitive. At the upper level the agent has a set of abstract actions $A_1, A_2, \ldots A_n$ which allows it to form plans to complete a range of useful tasks. These abstract actions could include simple things like picking up and putting down objects, or more complex actions such as using a cup to collect water from a tap. The aim of learning is to expand the set of useful abstract actions which are available to the agent.

Each abstract action can be used to generate a low-level behaviour by a process which we will outline in detail in Section 3.3.2. Abstract action goals (which describe many primitive states) are operationalised into a specific primitive world state through the use of a constraint solver. A motion planner and controller are then used to create the 'behaviour' which reaches the goal state.

3.3.1 Abstract action models

Each abstract action is represented by a model which consists of three pieces of information:

- `MOVING`: a list of manipulated objects
- `PRIMITIVES`: a list of movement primitives
- `STRIPS`: a STRIPS model (Fikes and Nilsson, 1971)

For the purposes of illustration consider the case of a robot arm using a cup to collect water from a tap. The act of placing the cup into a suitable pose under the tap can be represented as the abstract action `put_under(Cup,Tap)`. An abstract action model for this action might be:

```
ACTION        put_under(Cup,Tap)
MOVING        robot-arm, Cup
PRIMITIVES    fwd, back, left, right, up, down, rotatecw, rotateccw
STRIPS
              PRE  in_gripper(Cup),
                   gripping(Cup),
                   clear_underneath(Tap),
                   orientation(Cup,vertical-up)
              ADD  below( Cup, Tap),
                   near( Cup, Tap),
                   aligned_vertically( Cup, Tap),
              DEL  clear_underneath(Tap)
```

The list of manipulated objects, `MOVING`, in this case is simply the robot arm and the cup it is holding. The movement primitives, `PRIMITIVES`, are used by the low-level motion planner to generate solution paths. In this example they consist of 8 primitives which move the robot arm's end effector (and therefore the cup) in any of 8 different directions in real space.

The STRIPS model describes the conditions which must exist before the action can be executed, as well as the changes which occur in the world when the action is executed. STRIPS models are composed of three lists of abstract literals:

1. The precondition (`PRE`): A list of conditions which must be true in order for the action to be executable.
2. The add list (`ADD`): Literals which are added to the world state when the action is executed.
3. The delete list (`DEL`): Literals which are removed from the world state when the action is executed

In the above example of a robot arm using a cup, the STRIPS model precondition states that the cup must be in the arm's gripper, it must be oriented vertically, and the region underneath the tap must be clear of other objects. The add and delete lists (known as the *effects* of the action) state that the action results in the cup being located under the tap, as well as near the tap, and vertically aligned with the tap. A "side effect" of the action is the delete list element stating that the region below the tap is no longer clear.

As we shall discuss further below, STRIPS models play an important role in our action representation. They are used as planning operators in the agent's STRIPS planner, and they also provide the logical constraints from which low-level behaviours can be generated. An extension to our architecture to allow the use of richer action models is discussed in Chapter 7.

3.3.2 Generating a behaviour from an abstract action

The mechanism for generating a behaviour from an abstract action model is illustrated in Figure 3.3. There are five main steps involved:

1. **Extract the spatial sub-goals from the action's STRIPS effects.** Each predicate in agent's background knowledge is specified as spatial or non-spatial. Spatial predicates are expressed in terms of primitive poses or other spatial predicates.
2. **Generate a primitive goal state satisfying these sub-goals:** A spatial constraint solver is used to find a primitive world state which satisfies the abstract spatial subgoals of the action.
3. **Create a 'virtual' composite object to be manipuated by the motion planner:** Following our restriction to prehensile object manipulation

in this thesis (see Chapter 1), moveable objects listed in the `MOVING` component of the action model are treated as a single geometric object to be manipulated by the motion planner (joints in the robot manipulator are, of course, permitted). For example, a box and the agent pushing it are treated as a single rigid object for the purposes of motion planning. The relative spatial positions of the component objects are fixed to mirror their relative poses in the current world state. The allowed movement primitives are given by the `PRIMITIVES` list.

4. **Find a path to the primitive goal state:** A motion planner uses the composite object motion primitives to generate a path from the current world state to the ground goal state (generated in step 2).
5. **Track the generated path to the goal:** Track the path using a generic feedback controller, following it until the desired primitive goal state is reached. Failures are detected by a simple timeout mechanism and handled at the abstract level by replanning.

In the case of the example of a robot arm using a cup, the agent extracts the spatial subgoals from the action model and generates a concrete goal state in which the cup is positioned correctly under the tap. It then constructs a composite object for the motion planner by attaching the cup to the robot arm's end effector. The allowed movements of the arm-plus-cup combination are the same as the default set of movement primitives for the arm alone. The composite object and movement primitives are passed to a motion planner which finds a path from the current world state, to the goal state in which the cup is being held under the tap. A generic controller is then used to track this path and move the cup to this location.

Further description of the ground goal state generation step are given below, while in-depth details on the constraint solver, path planner, and controller used in our agent are presented in Chapter 6.

Generating a ground goal state

Abstract goals are converted into a ground goal state using a constraint solver which we have written in the constraint logic programming language ECLiPSe

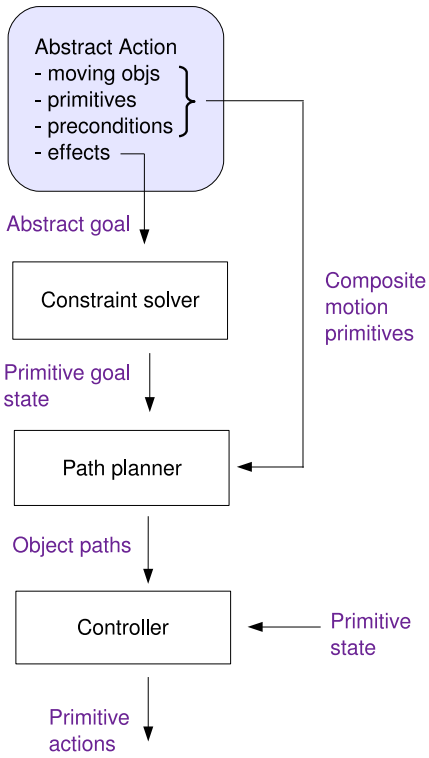


Figure 3.3: Illustration of how behaviours are generated in our agent architecture. The action model's moving object list, movement primitives, and preconditions define a set of composite object motion primitives which can be input into a path planner. The abstract goals of the action are extracted from the STRIPS effects and passed to a constraint solver, which produces a ground goal state. The path planner finds a path to move objects to the goal state, and the generated behaviour is produced by a controller tracking this path to achieve the desired (ground) goal state and abstract action sub-goals.

(Apt and Wallace, 2007). The details of how this constraint solver works are presented in Chapter 6 so here we simply give a brief overview of how it works in practice.

Firstly it should be noted that the constraint solver only operates on spatial predicates. Thus, given the literal `on(book,table)` as a goal the constraint solver is able to generate a ground world state in which the book is on the table. However non-spatial literals such as `painted(table)` are ignored: any non-spatial effects are treated as side-effects of the movement and interaction of objects in the world (such as a wet paint brush moving across the surface of the table).

The constraint solver takes the following inputs:

- a set of abstract spatial constraints
- a composite "moveable" object comprised of one or more bodies (often in a fixed relative spatial orientation)
- a "background" state of objects in fixed poses

In the arm and cup example, the input abstract spatial constraints would be (from the STRIPS model): {`below(cup,tap)`, `near(cup,tap)`, `aligned_vertically(cup,tap)`}. The composite moveable object would be the robot arm, with the cup attached to the end effector (in a position mirroring the current world state). Finally, the background state would consist of the current poses of all other objects in the vicinity – excluding the arm and the cup – and these objects are treated as fixed in space.

The constraint solver treats the poses of the bodies in the composite moveable object (eg. the arm and cup) as variables, and searches for solution poses for these objects such that the abstract spatial constraints are satisfied. The solution search space therefore consists of the space of allowed (non-colliding) poses of the composite moveable object. The algorithm which does the low-level constraint satisfaction search is built-in to the ECLiPSe engine, and optimised to search efficiently through the space without needing to enumerate all of the possible poses.

In most cases a variety of ground pose solutions for the composite object are possible and a particular solution is chosen at random. For example, when solving a constraint such as `on(book,table)` the solution pose for the book could lie anywhere on the table's surface. Additional spatial constraints would be required if a specific location on the table was desired.

3.3.3 Manipulation recognition models

In addition to a 'library' of abstract actions, the agent also stores a set of *manipulation recognition models*. A manipulation recognition model is a special type of abstract action model whose STRIPS model contains no effects. It is not used

for planning or generating actions, but for recognising particular types of actions executed by other agents.

As an example, consider the following manipulation recognition model for moving an object to a new location:

```
ACTION   move_object(Object)
MOVING   robot-arm, Object
STRIPS
  PRE    in_gripper(Object),
         gripping(Object)
  ADD    -
  DEL    -
```

This manipulation recognition model would allow an agent to recognise any action which involved moving an object to a new location when the object is being held by the robot's gripper. As we shall see in Chapter 4, the absence of effects in the manipulation model will allow it to be matched to segments of the teacher's demonstration where the type of object manipulation is known, but the intended goal of the manipulation is not. This situation occurs frequently in tool-use problems, where a single type of action can be used to achieve many different abstract goals.

As a further example, suppose the learner is watching a teacher push a box to a position under a light bulb, in order to climb up on it and change the bulb. If the learner is unfamiliar with using boxes to increase one's reach it will not recognise the significance of the state `under(box,lightbulb)` which defines the goal of the pushing action. However a "pushing" recognition model would still be able to recognise the type of action which is occurring (which means the learner can focus on learning the action sub-goal).

3.3.4 Discussion

Before moving on to the presentation of the agent's learning algorithms in Chapter 4, it is useful to briefly comment on the role of STRIPS models in our representa-

tion and learning.

Our STRIPS models have two primary uses: firstly as traditional planning operators, and secondly as generators of low-level behaviours. We say that our STRIPS operators "generate" low-level behaviours because when the sub-goals of the action (encoded in the STRIPS effects) are changed, the observed low-level behaviour of the agent changes. Thus by learning desirable action sub-goals and preconditions (which are themselves sub-goals) we are learning different ways of acting. An alternative way of looking at it is to say that we are trying to identify useful instances of a single generic motion-planner-plus-controller behaviour (which can accept multiple objects as inputs).

Chapter 4

Learning

In this chapter we describe the algorithms used by the agent to learn new tool-based actions. Our approach is divided into two parts:

- learning by explanation
- learning by trial and error

In the first part, our agent tries to identify a new and useful tool-use action by watching the activities of a teacher. An explanation-based learning method is used to construct an approximate action model representing the novel tool action.

In the second part, the agent attempts to generalise this new tool action through trial and error experimentation. This involves selecting potential tool objects and testing out ways of using them to achieve the goal. By incrementally refining and generalising its hypothesis, the agent is able to learn both the spatial relationships and the object properties required for successful tool use.

4.1 Assumptions: Structure of a tool use action

Most tool actions consist of two or more component steps. Consider the following common tool actions for example:

- placing a ladder against a wall and then climbing it
- putting a bucket under a tap, then turning a tap to fill it with water
- placing a nail and then hitting it with a hammer

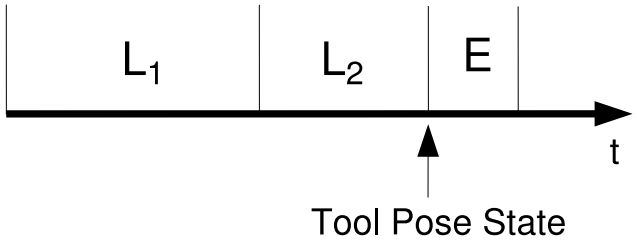


Figure 4.1: Illustration of the "tool pose state". The tool action has the structure $L_1, L_2, \ldots, L_n, E$ where L_i denotes a positioning step, and E an effects step.

- placing the blade of a peeler onto a carrot, then dragging it across the carrot's surface
- putting paper in a stapler, placing a hand on the stapler, and pushing downwards

A common feature of these actions, and many others, is that the action can be divided into two main parts:

- **The positioning step(s)**: The agent moves the tool and/or target objects into a correct relative pose.
- **The effect step**: The agent applies the tool (executes a primitive action or behaviour) to achieve a useful effect.

If we denote a positioning step by L_i and the effect step by E then a typical tool action with n positioning steps has the structure: $L_1, L_2, \ldots, L_n, E$.

Definition: The tool pose state

Actions which fit this pattern have an important property which greatly simplifies the learning process: all of the interesting spatial information about using the tool is encapsulated in the world state which immediately precedes the effect step. This world state, which we shall call the **tool pose state**, is illustrated in Figure 4.1.

The tool pose state represents a snapshot of the robot, tool, and relevant objects in their correct relative positions ready for use. Examples of the tool pose state are:

- a bucket sitting underneath a tap, with the robot's hand on the tap

- a carrot peeler being held with its blade against the carrot, at right angles to the length of the carrot
- a screw attached to a drill, and being held up against a piece of wood

In each case the tool pose state features the tool, objects, and agent in a relative spatial pose which is characteristic of the tool action being executed. Our aim is for the agent to learn a description of this state so that it can reproduce the tool action.

Learnable tool actions

In this thesis we assume that the tool actions which the agent must learn correspond to the above "positioning/effect" pattern. This domain restriction allows an important simplification in the learning process. Specifically, the learner only has to learn to describe a single tool pose state, rather than learning a *sequence* of intermediate sub-goal states.

Learning a sequence of intermediate pose states is a significantly more difficult problem, since the agent only receives feedback about the success or failure of the whole sequence (rather than the individual components). When an action fails it is then difficult to work out which sub-goal state was incorrectly defined. In the general case, where the agent must pass through many intermediate sub-states before getting a succeeded/failed signal, the problem turns into (relational) reinforcement learning. In this context, working out which intermediate states were important is the so-called credit assignment problem.

A simple example of an action with multiple intermediate pose states is painting a wall: the agent must first correctly place the paintbrush in the tin of paint (tool pose state 1), before placing the brush on the wall (tool pose state 2). The success of the painting action depends on passing through *both* intermediate tool pose states. If the final painting action fails, then it is difficult to know which intermediate state was incorrect.

One way around this problem is to receive many more examples of the correct behaviour from the teacher. This allows the learner to isolate the important features of the intermediate states it must pass through. Learning (relational)

action sequences from multiple teacher examples has been studied previously (see eg. (Benson, 1996; Hume, 1995)) and we do not pursue it further here — our goal in this thesis is to demonstrate fast learning from a single example of tool use.

4.2 Learning from explanation

In this section we describe how the agent is able to identify novel tool actions in a teacher's demonstration of a tool use task. We present an explanation-based method for constructing a new abstract tool action model. This model is later used as the starting point for trial and error learning in Section 4.3.

The problem faced by the learner is illustrated in Figure 4.2. It sees a teacher performing an activity which leads to a useful goal being achieved. The learner recognises some of the actions it sees (shown in green), but somewhere in the middle of the demonstration is a novel action it does not recognise. The aim is to explain what the novel action does, and to build an initial abstract model of this action.

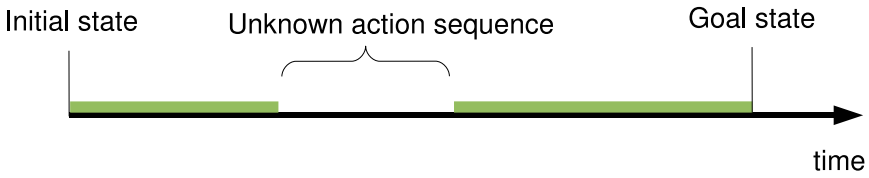


Figure 4.2: Illustration of an unrecognised action occurring during the teacher's demonstration. The green line represents the parts of the demonstration which were recognised by the learner.

Our approach consists of the following steps, illustrated in Figure 4.3

1. **Watch a teacher using a tool to complete a task.**
2. **Identify abstract actions in the teacher's demonstration.** This consists of two steps:
   (a) **Segment the demonstration.** The agent divides the teacher's demonstration into segments, each corresponding to a different unlabelled abstract action.

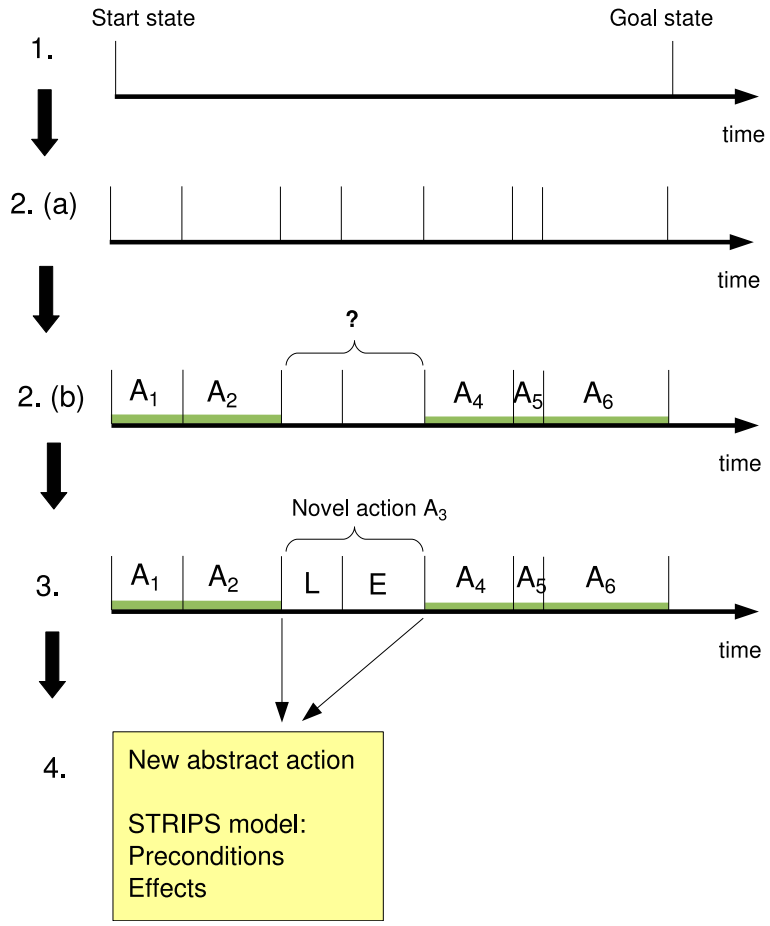


Figure 4.3: Summary of learning a new action model by explanation. The steps involved are: 1. Watch the teacher's demonstration; 2. (a) Segment the demonstration; (b) Match the segments to existing action models; 3. Create a new action to represent the unknown segments; 4. Build a corresponding STRIPS model by explanation.

(b) **Match the segments to abstract actions.** The agent attempts to label the segments of the demonstration using the actions in its existing library of abstract actions.

3. **Introduce a novel tool action to represent unrecognised segments.** Any segments which cannot be matched to known actions are labelled as components of a novel action.
4. **Construct a novel action model via a form of explanation-based**

learning. A STRIPS model for the novel action can be constructed by identifying the subset of literals in the novel action's start and end states which are relevant to explaining how the teacher achieved the goal.

5. **Pass the new action model to the trial-and-error learning module for further refinement**. The process for learning by experimentation is described in Section 4.3. It involves testing various tool objects on related problems in order to learn a better model of the new action.

Each of these steps are discussed in more detail in the remainder of this section.

4.2.1 Segmentation of the teacher's demonstration

The learner is given a demonstration of the novel tool use task by a teacher. This demonstration is supplied to the learner in the form of a sequence of primitive world states $w_1, w_2, w_3, \ldots, w_n$, where each primitive world state w_i specifies the poses of all objects in the world at time step i. The primitive world state sequence is created by sampling the world every 0.1 seconds during the demonstration.

In order to identify the abstract actions executed by the teacher agent the learner uses a two-step process. Firstly the demonstration is segmented using heuristics described below. These segments are then matched to abstract action models in the agent's background knowledge. The model matching process is described in Section 4.2.2.

A simple example

We use two simple heuristics to identify boundaries for segmentation of a teacher's example. Firstly, the **object motion heuristic** which states that a distinct action begins or ends each time an object (or the agent) starts or stops moving. Secondly the **object contact heuristic** states that actions may also start or stop when two objects come into contact or break contact.

As an illustration of segmentation, consider a teacher using a broom to retrieve a box which is out of reach under a couch. We might naturally describe the activity as follows: the agent picks up the broom, it puts the broom under the couch and

hooks the box, then pulls the box out from under the couch; finally it puts the broom down and picks up the box.

This natural segmentation obeys the object motion heuristic: in each segment, a different combination of objects is moving. This idea is illustrated in Table 4.1 below:

Segment	Teacher's action	Objects in motion
1	Reach to pick up broom	agent
2	Move broom under couch	agent, broom
3	Pull box from under couch with broom	agent, broom, box
4	Put broom down	agent, broom
5	Reach to pick up box	agent
6	Lift box	agent, box

Table 4.1: Movement-based segmentation of the broom problem.

This segmentation can then be used for matching abstract actions to the demonstration trace. An abstract action matches a segment if it has the same set of moving objects, and if the preconditions and effects of the STRIPS model are satisfied by the start and end points of the segment. This matching process is discussed in Section 4.2.2.

Why segmentation?

Before presenting the details of the segmentation algorithm we should comment briefly on the reason for separating the segmentation and action model matching steps. Why don't we simply try and match STRIPS models to different parts of the demonstration trace and skip the motion/contact segmentation step altogether?

The answer is threefold. Firstly, the novel tool action frequently involves multiple unrecognised steps. For example in the broom example described above, neither segments 2 or 3 (hooking the box with the broom, and then pulling the box) would be recognised by the learner. A purely STRIPS-based approach would lump these two steps together as a single "unrecognised" action, and provide no clues as to how it should be divided into sensible components (if at all).

Secondly, matching at only the STRIPS level can introduce a great deal of

ambiguity as to when particular actions begin and end. This is because an action's precondition can be applicable long before it is actually executed (see Figure 4.4).

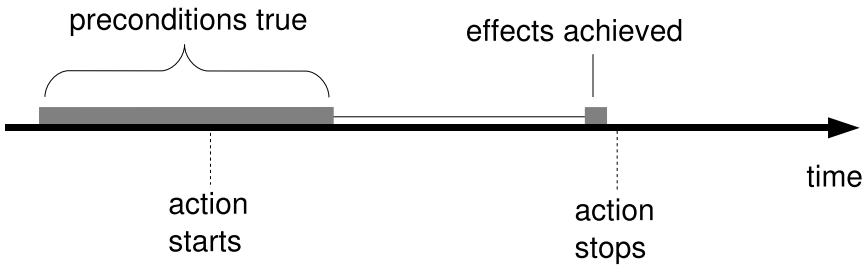


Figure 4.4: Recognition of actions using STRIPS models alone can be problematic, since the action preconditions give only limited information about when the action actually commenced.

Thirdly, our approach ensures that the start and end points of all components of the tool use action can be identified clearly. Since we use matching at both the primitive movement/contact level, and at the abstract STRIPS level, our novel action recognition system also makes it more likely the agent will match known actions to observations correctly.

Motion-based segmentation

The primary action segmentation heuristic used in our work is:

> **The object motion heuristic**: *Action segments begin or end when objects start or stop moving.*

Many human and robot activities, including tool use, can be naturally segmented into distinct actions using this heuristic. The heuristic is appealing from a robotics viewpoint because it relies only on being able to detect when objects start or stop moving. Using motion to broadly segment activities has been used a number of times in others' work, such as (Sridhar et al., 2008) (as discussed in Chapter 2).

We apply motion-based segmentation through a two-step process. Firstly, the velocity of each object is thresholded to produce motion boundaries for each object individually. These individual object boundaries are then combined into a single segmentation of the teacher's trace.

The idea is illustrated in Figure 4.5, which shows how the segmentation process would work for the broom problem described earlier in this chapter. The observed velocities of the robot arm (end effector), the broom, and the box are first thresholded individually. Combining these boundaries produces six segments corresponding to those listed in Table 4.1 presented earlier. Segment three, for example, involves the robot manipulating the broom and the box.

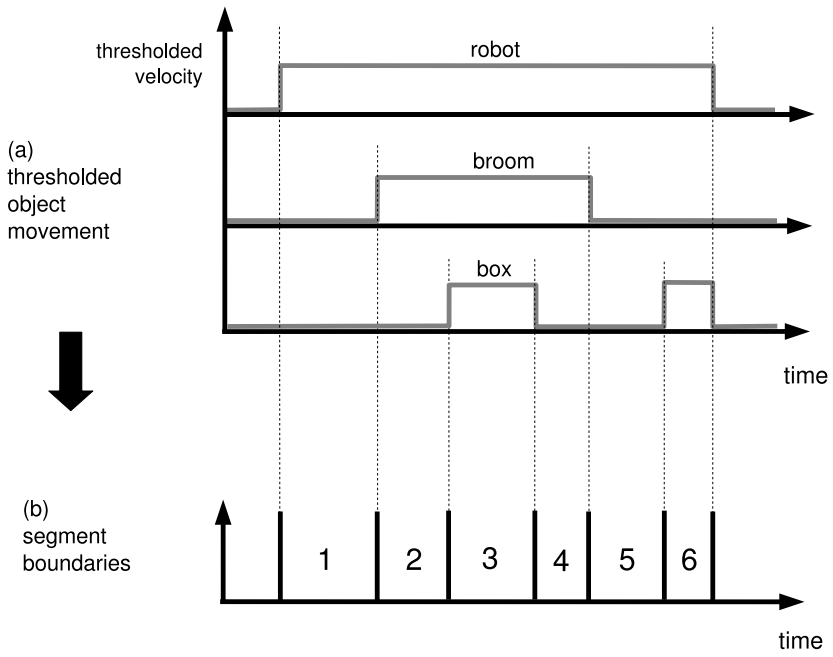


Figure 4.5: Segmentation of the broom problem using the object motion heuristic. The numbered segments correspond to those in Table 4.1.

We use a "hysteresis" threshold for thresholding the individual object velocities — that is, we use different thresholds for starting and stopping movement such that $v_{start} > v_{stop}$. This ensures that objects do not appear to "flicker" between the moving and stationary states. The exact values of the v_{start} and v_{stop} parameters must be set by the user, but the system is not terribly sensitive to the exact values chosen.

A further complication may arise from cases where the teacher robot pauses during execution of the example. For example, the teacher may move to a particular location, stop, turn, and then continue moving in a new direction. Since

nothing is happening during the "paused" segments we simply ignore boundaries created during paused sections of the demonstration trace. Similarly, very short segments created by objects starting or stopping motion very briefly are filtered out.

Lastly, it is often the case that two objects will start or stop moving at roughly – but not exactly – the same time. This situation is illustrated in Figure 4.6. To avoid producing two very closely spaced boundaries in the combined trace we merge segments which fall within a short time t_{merge} of each other.

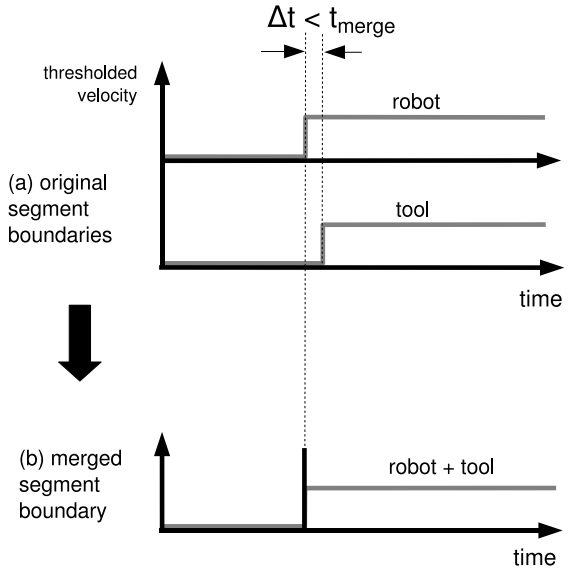


Figure 4.6: Merging segment boundaries: Boundaries are merged if they fall within $\Delta t < t_{merge}$ of each other.

Contact-based segmentation

A second useful heuristic for segmenting activities is one we refer to as the contact heuristic:

> **The contact heuristic**: *Action segments begin or end when objects come into contact or break contact.*

This heuristic generally produces segment boundaries in similar locations to the object motion heuristic — the reason being that objects usually start or stop

moving as a result of contact with another body.

However the heuristic is useful in segmenting activities which involve "applying" a moving object to a stationary object: buttering a piece of bread with a knife or wiping a blackboard with a cloth, for example. In these situations the tool object moves across the surface of another object and it is useful to be able to isolate the component of the action corresponding to this motion. The object motion heuristic would not recognise these as separate segments as only the tool object moves.

The heuristic is applied in a similar manner to the object motion heuristic, except that it is applied *within* existing motion segments. That is, we only attempt to find contact segments within existing motion segments. For each motion segment we step through the demonstration trace and note any points at which objects make or break contact — "contact" is defined by upper and lower distance thresholds between objects. Contact boundaries that are close to the beginning or end of the existing segment are ignored, since they are already accounted for. In many cases this process does not produce any new segments, but in problems such as those described above one or more new segments are introduced.

The reason for not using *only* the contact heuristic (and ignoring motion segmentation) is that there can be some difficultly in determining the point at which objects actually come into contact. This is particularly true in a robotic's context, where determining whether an object is moving in a camera image is a simpler task than determining whether two objects are in contact. We therefore treat motion segmentation as the primary heuristic.

4.2.2 Matching segments to abstract actions

Having divided the teacher's example into distinct segments, the agent attempts to match them to actions in its action library. A segment is matched with an abstract action A_i from the agent's library if the following conditions are satisfied:

1. The objects manipulated in the segment can be matched to the `MOVING` list in the abstract action model of A_i
2. The preconditions of A_i are true at the beginning of the segment

3. The effects of A_i are true at the end of the segment

In the (unusual) case where more than one abstract action can be matched to a particular segment, the segment is labelled with the action with the more specific set of preconditions.

If the segment cannot be matched to a known abstract action, the learner checks to see whether it matches with a manipulation recognition model (defined in Chapter 3). Recall from Section 3.3.3 that manipulation recognition models consist of only a set of preconditions and a set of moving objects. As the name suggests, these models allow the agent to identify the manner in which an object is being moved even if the abstract goal of the action is unknown. For example, different manipulation models might detect an object being held in a gripper, an object being pushed in front of the robot, or an object being carried on top of another object.

In the broom problem described earlier, the learner's manipulation model can recognise that the teacher is holding the broom and moving it to a new position, even if it does not recognise the intended goal state for the tool pose.

Finally, if the segment cannot be matched to either an abstract action or a manipulation recognition model, then it is simply labelled as unknown. These types of actions involve objects and/or tools being manipulated in novel ways, since the recogniser models summarise the methods of manipulation which the agent already knows about. Novel manipulations usually occur in the "effect step" of the action, such as in peeling a carrot or using a screwdriver to turn a screw. In the broom problem the agent does not recognise the type of manipulation in the "pulling" segment where the broom drags the box from under the couch, and thus this segment is labelled as unknown.

The segments which are not matched to a known abstract action A_i are (following the assumptions in Section 4.1) grouped together as components of a compound tool action. This typically comprises a sequence of positioning steps which have been labelled with manipulation recognition models, and an effects step which has an unknown manipulation type (ie. is unmatched).

4.2.3 Explanation-based learning of the STRIPS model

The learner now has a segmentation of the teacher's demonstration, with each segment either labelled by a known action or labelled as part of a novel action. The next step is to construct an abstract model of the novel action which defines how it is executed and what it achieves. Below we present an explanation-based approach to constructing this action model, which relies on examining the context in which the action is executed to determine relevant preconditions and effects. Our approach is a form of explanation-based learning (EBL) but it differs from the original concept which focused mostly on speed-up learning (Mitchell et al., 1986).

The explanation-based heuristic we use is based upon the assumption that the teacher is acting rationally, and states that each action in the demonstration sequence is executed in order to achieve a necessary sub-goal. Therefore actions occurring before the novel action should enable the novel action preconditions. Similarly, the effects of the novel action should help enable the preconditions of actions occurring later in the demonstration.

To formalise this idea, let us firstly define *unsupported preconditions* as preconditions which are not enabled by the (known) effects of any preceeding actions (and which were not already true at the start of the demonstration). We also define *unexplained effects* as effects which occur in an action segment but are not explained by the corresponding action model (if it exists), *and* which enable the preconditions of one or more actions occurring later in the explanation.

If we assume the teacher is acting in a rational manner then any explanation of the plan it has executed must obey the following properties:

- no action in the explanation should have unsupported preconditions
- there should be no unexplained effects which are not accounted for in the explanation

These requirements can be used to infer some of the missing preconditions and effects of the novel action segments. The algorithm used to build an explanation-based model of the novel action segments is as follows:

1. **Identify unsupported preconditions** in action segments occurring after the novel action.
2. **Identify unexplained effects** occurring in the novel action segment, which can be used to explain how the unsupported preconditions were enabled.
3. **Construct new action models** which account for the unexplained effects and enable the unsupported preconditions in the explanation:
   - **The "effects"-step** STRIPS model is defined as:
     - **Effects**: Defined as the intersection of the observed state changes which occur during the segment and the unsupported preconditions of actions occurring after the novel action.
     - **Preconditions**: Defined as the net effects of previous actions, plus a literal `tool_pose(Tool,Obj)` representing the tool pose state which must be achieved in order to enable the action. The definition of the `tool_pose` literal must be learnt through trial-and-error experimentation (Section 4.3).
   - **A "positioning"-step** STRIPS model is defined as:
     - **Effects**: Defined as any unexplained effects occurring during the segment *plus* an additional `tool_pose(Tool,Obj)` literal. The `tool_pose(Tool,Obj)` represents the tool pose state which must be achieved in order to enable the effects-step of the tool action defined above.
     - **Preconditions**: The preconditions are copied from the recognition model which was matched to the segment.
4. **Replace ground objects in the action model by variables.** We only replace constants with variables if they refer to physical objects; other types of logical constants are preserved.
5. **Assign the novel action a unique name and define the action parameters.** The parameters are defined to be the list of objects moved by the action (excluding the robot), along with any other variables found in the preconditions or effects list.

As an example of the output of this algorithm, consider once again the broom problem described in Section 4.2.1. The learner watches a demonstration and

segments it into the six segments shown in Table 4.2. In this table Segment 2 is recognised as a positioning action (it matched a manipulation recognition model) whilst segment 3 was unrecognised. These two segments are components of a novel tool action.

Seg	Action	Unexplained effect	Unsupported precon
1	Pickup broom	-	-
2	?Positioning?	-	-
3	?Unknown?	¬`under(box,couch)`	-
4	Put broom down	-	-
5	Pickup box	-	¬`under(box,couch)`
6	Lift box	-	-

Table 4.2: Unexplained effects and unsupported preconditions in the broom problem.

The algorithm checks to see if there are any unsupported preconditions in the explanation occurring in segments after the novel action and finds that ¬`under(box, couch)` is an unsupported precondition of `pickup(box)` — that is, there are no existing actions in the explanation which can account for how this precondition was enabled.

The algorithm then checks to see whether the novel tool action segments are able to explain how this precondition was supported. It finds that the effect ¬`under(box,couch)` occurs during segment 4. The action model for this segment must therefore have ¬`under(box,couch)` as an effect. Note that in general there will be many irrelevant effects which occur when an action is executed. We use the context in which the action is executed (the explanation) to help identify the small number of effects which are relevant. This situation is illustrated in Figure 4.7.

In order to construct the corresponding preconditions of the tool action, we make an assumption that any literal which is an effect of an earlier action in the explanation, and is still true at the time of execution of the action, is likely to be an action precondition. In addition to these "explained" preconditions, we also add a `tool_pose` literal to the precondition — which represents the definition of the tool pose state which will be learnt by trial-and-error in Section 4.3. This

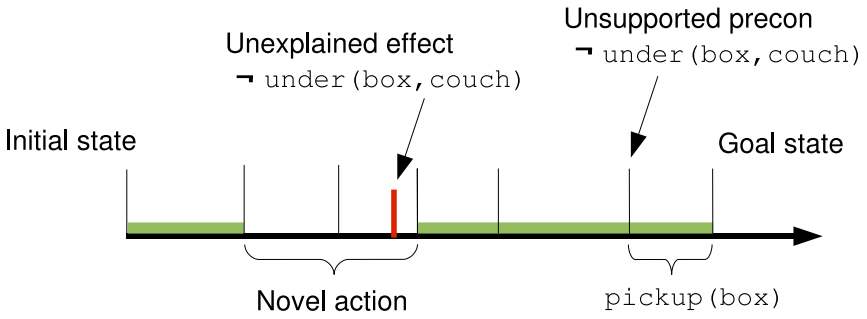


Figure 4.7: Illustration of an unsupported precondition and corresponding unexplained effect in the broom problem. Unsupported preconditions in the teacher's explanation are used to identify the important effects of a novel action, as in general many of them are irrelevant.

derived predicate describes the spatial positioning of the tool, and the necessary properties of the tool. Following this algorithm, and after converting constants to variables the constructed action model for using the broom to get the box is as follows:

```
drag-with-broom(Broom,Object,Couch):
PRE:   under(Object,Couch),
       holding(Broom)
       tool_pose(Broom,Object)
ADD:   -
DEL:   under(Object,Couch)
```

The positioning-step component of the tool action is constructed in a similar way, except that the `tool_pose` literal is added as an effect of the action, so that this action becomes necessary in order to enable the effect-step of the action. The action model constructed in this case is:

```
position-broom(Broom,Object):
PRE:   holding(Broom)
ADD:   tool_pose(Broom,Object)
DEL:   -
```

The explanation-based learning approach we have described here relies on the assumption that the learner's existing action models are sufficiently complete to describe the teacher's demonstration (excluding the novel action). The method will fail if, for example, the agent does not know that ¬`under(box,couch)` is a precondition for picking up the box. However in this situation the agent would have no reason to learn the new tool action, because according to its own action models it should be able to pick the object up directly. Of course, if the agent were to attempt to do so it could then learn that its existing action models are incorrect, and that ¬`under(box,couch)` should be part of the precondition.

In this thesis we focus on learning tool actions which produce *useful* effects — that is, effects which enable one or more preconditions in the agent's existing action models. Actions which produce effects that do not help enable any other action are irrelevant as far as a planner is concerned.

4.3 Learning by trial and error

After explaining the teacher's demonstration the agent has an initial action model to represent the novel tool action. The learner must now refine this model through trial-and-error experimentation. It does this by learning a definition of the `tool_pose` predicate which appears in the precondition of the action. As we have noted, this predicate is used to represent the required preconditions of the tool pose state which were not learnt by explanation of the teacher's demonstration. The definition of `tool_pose` should include any additional preconditions which describe the required spatial positioning, structure, or shape of the tool.

Our agent learns a definition of the tool pose state by treating it as a form of concept learning problem in which examples must be generated through experimentation. Each time the action is executed, the agent notes the tool pose state from which the action was executed and whether the desired sub-goals of the action were achieved. If the action completed successfully then the state is a positive example of the correct tool pose state concept; conversely, if the action fails to achieve its subgoals then the recorded state is a negative example. This concept learning approach is commonly used in action model learning (see eg. (Benson,

1996; Pasula et al., 2004)).

We define the tool pose state `tool_pose(Tool,Obj)` (which appears in the tool action precondition) as the learning concept and then repeatedly:

1. **Test the current hypothesis:** Select a tool which satisfies the hypothesis and place it in a pose defined by the spatial constraints in the hypothesis. Generate a motion plan which solves the task from this state and execute it, observing whether the action sub-goal is achieved.
2. **Add a new positive or negative example:** If the action achieved the desired sub-goal, label the initial state as a positive example. If the action failed to achieve the desired sub-goal, label it as a negative example.
3. **Update the hypothesis:** Run a relational concept learner to update the definition of correct tool pose state.
4. **Generate a new learning task, or reset the current one:** If the current task was successfully solved, a new learning task is generated and presented to the agent. If the agent failed then the current task is reset.
5. **Repeat:** The agent continues its experimentation on a sequence of learning tasks, refining its hypothesis. The experiment is terminated when the agent is able to solve a pre-defined number of consecutive tasks without failure.

The usual approach to using a relational concept learner to learn action model definitions involves batch learning (eg. (Pasula et al., 2004)), where actions are executed many times, and the concept learner run only after all the examples have been collected. We use an incremental approach, which allows the hypothesis to be refined after each example. The benefits include:

- fewer examples, since the exploration is more targeted
- the agent can act to "exploit" its learning early on (from the very first example in many cases)

In the remainder of this section we present the details of how tasks are generated, the hypothesis is represented, examples are generated, and of course the generalisation algorithm itself.

4.3.1 Generation of learning tasks

The learner is presented with a succession of learning tasks. Each task features a different set of available tool objects (random variation), and a variation on the arrangement and dimensions of the other objects in the world. For example, each instance of a "hammering" task would involve a different set of available hammering tools along with differently sized or shaped nails.

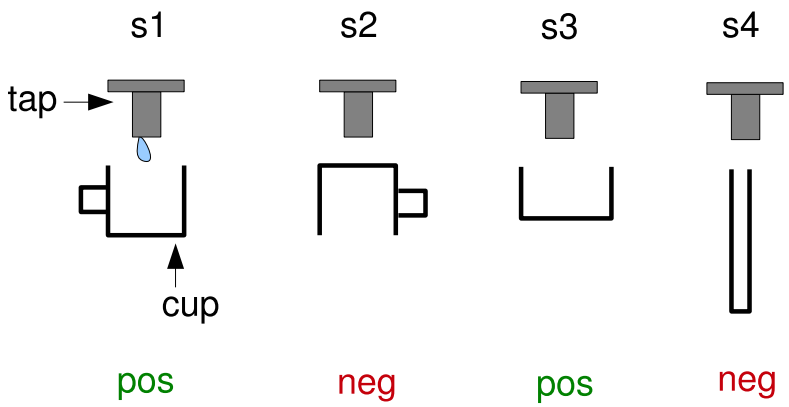
The details of how we represent the task definition which generates each random instance are described in Chapter 6. For now, we simply note that the agent is presented with a new, randomly-generated task instance each time it solves the previous task. If the agent fails to solve the task by using a particular tool, the world state is reset so that it may attempt the same task again (using a different tool, or a different tool pose state). This "reset" step is not strictly necessary from the point of view of our learning algorithm, but is performed in order to speed up the experimentation process. Some previous action model learners (Hume, 1995) have focused on action learning tasks where a world reset is not possible.

4.3.2 Representation of examples

The learning examples in our problem are positive and negative instances of the tool pose state (see Section 4.1 for the definition of this state). This state is a precondition of the "effect step" of the tool action. It determines both the correct positioning of the tool and the necessary properties of the tool (dimensions, shape, geometric construction and so on).

Figure 4.8 shows four examples of the tool pose state for the task of using a cup to collect water. Positive examples require that the tool (cup) be placed in the correct spatial pose, and also that it has the correct structure. The second example in the figure shows a correct "cup" structure but in the wrong orientation. The fourth example shows a permissible spatial pose but the structure of the tool is wrong (it is too thin).

Examples of the tool pose state are represented as instances of the `tool_pose` predicate. For example, the third cup-use example is represented by the predicate `tool_pose(cup2,s3)`. The parameter $s3$ appearing in this predicate is the state



State label	Example	Type
s1	tool_pose(cup3, s1)	pos
s2	tool_pose(cup3, s2)	neg
s3	tool_pose(cup2, s3)	pos
s4	tool_pose(cup7, s4)	neg

Figure 4.8: Positive and negative examples of correct "cup" tool use. Each state is labelled by s_n and the example by `tool_pose(`*Tool*`,`*State*`)`. Note that the position of the robot's gripper is not shown in this simple illustration.

label — the nth example state is assigned the label s_n. The learner records whether an example is a positive or negative example of correct tool use by simply writing (for state $s3$ in the Figure):

```
example( tool_pose(cup2,s3), pos).
```

Whilst `tool_pose(Tool,State)` provides a label for an example, the full specification of the example must include all of the abstract literals (also referred to as *facts*) which describe the state. Any abstract literal which is true in the tool pose state is included in the list of facts associated with an example. For instance, the state $s2$ would include the following facts (amongst others):

```
orientation(cup3,upsidedown,s2).
on_axis(cup3,tap,s2).
under(cup3,tap,s2).
```

These facts describe the spatial positioning of the tool relative to other objects in

the world. Note that a state parameter is included as the last parameter of each fact.

Finally, in addition to the dynamic spatial facts associated with an example there are also a set of "static" structural facts, which describe the structure of each object. Since the structure of an object does not change (at least in our work) an additional state parameter is unnecessary. As an illustration, some simple structural facts describing state $s4$ might be:

```
narrower(cup7,tap).
taller(cup7,tap).
part_of(cup7,cup7_leftwall).
part_of(cup7,cup7_rightwall).
parallel(cup7_leftwall,cup7_rightwall).
```

These static background facts are stored alongside the dynamic spatial facts for each example.

4.3.3 Representation of the hypothesis

As we have seen, examples of correct tool use are represented by ground instances of the predicate `tool_pose(Tool,State)`. The agent's hypothesis describing correct tool use can then be represented by a clause with `tool_pose(Tool,State)` as its head. For instance, a hypothesis which states that all cups must be `concave_up` in shape and held under a tap would be written:

```
tool_pose(Cup,State) :- in_gripper(Cup,State),
                        is_tap(Tap),
                        under(Cup,Tap,State),
                        shape(Cup,concave_up).
```

In this thesis, rather than keeping track of a single hypothesis clause and refining it incrementally, we instead maintain a set of of allowed hypotheses. We do this by keeping a single most-specific clause representing one bound on the hypothesis, and a single most-general clause representing the other. This can be viewed as a form of version space (Mitchell, 1978), though a true version space

keeps track of more than one hypothesis on the boundary. As we shall see, our most-general boundary clause is simply a reduction (a subset) of our most-specific boundary clause.

Our 'version space' of allowed hypotheses is illustrated in Figure 4.9. The outermost boundary on the space represents the shortest, most-general hypothesis clause consistent with the observed examples. The innermost boundary on the space represents the longest, most-specific hypothesis clause consistent with the examples. We denote the most-specific and most-general hypothesis boundaries by h_S and h_G respectively.

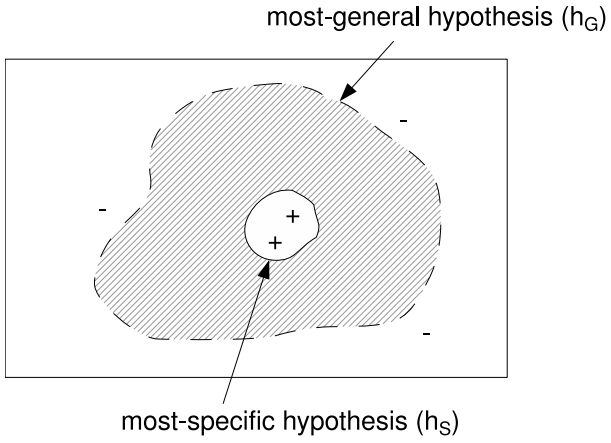


Figure 4.9: The most-specific (h_S) and most-general (h_G) boundaries on the hypothesis space. The + and − symbols in the figure represent positive and negative examples respectively.

This representation differs from that used in most ILP concept learners (eg. Progol (Muggleton, 1995)) which maintain a single most-general hypothesis clause and search in a general-to-specific manner. The difficulty with a general-to-specific approach in the tool-use domain is that the concepts to be learnt are usually described by fairly long clauses. Learning a lengthy clause in a general-to-specific manner is very inefficient — it requires very many examples due to the large branching factor and the fact that all possible shorter clauses must be enumerated before longer ones.

Similarly, an approach based on maintaining only a most-specific hypothesis would also suffer from disadvantages — the biggest difficulty being that it becomes

difficult to find examples which satisfy the current hypothesis. Usually only a subset of the literals in the most-specific hypothesis can be satisfied in the current state, and it is difficult to know which are most relevant. Our approach maintains the most-general boundary to help with this process.

We should note that we are not the first to use a form of version-space representation for learning action models. Wang's OBSERVER system (Wang, 1995) used a true version space (and so was not robust to noise) and Amir (2005) uses a more complex form of version-space filtering in partially observable domains.

Our motivation for keeping a most-specific and a most-general boundary on the hypothesis space is related to the efficiency of exploration in the tool-use domain. This point is elaborated in Section 4.3.4 where we demonstrate how both boundaries are used to find a suitable example to test.

Mode declarations

Following the approach of Progol (Muggleton, 1995), we constrain the search space for hypotheses by providing the learner with a list of mode declarations as background knowledge. Mode declarations are statements which restrict the ways in which a new literal can be combined in a hypothesis clause. Each mode declaration names a predicate which can appear in the body of the hypothesis clause, and restricts the way in which its parameters must be linked to existing parameters in the clause. For example the mode declarations:

```
:- modeb( near(+obj,-obj,+state)).
:- modeb( shorter(+obj,+obj)).
```

state that predicates `near(A,B,C)` and `shorter(A,B)` can appear in the body of the hypothesis clause.

Each parameter in the mode declaration consists of two parts: an input (+) or output specifier (−), followed by a type. The type simply requires the parameter to be of the named variety (eg. an `obj` or a `state`). An input specifier (+) means that the parameter must be identical to one which occurs earlier in the clause; an output specifier (−) allows it to be a new parameter which does not occur earlier in the clause.

Using these mode declarations, the following two clauses would be valid hypotheses:

```
tool_pose(A,State) :- near(A,B,State).
tool_pose(A,State) :- near(A,B,State), shorter(A,B).
```

whereas this clause would not:

```
tool_pose(A,State) :- shorter(A,B).
```

This clause is invalid according to the mode declarations because the second parameter of `shorter(A,B)` is defined as an input parameter but does not occur earlier in the clause.

4.3.4 Testing a hypothesis

There are essentially three ways of choosing a hypothesis from within the version space to test. The choices are:

- the most-specific boundary; or
- the most-general bounary; or
- a hypothesis in-between these boundaries

Our approach takes the third option of searching for a suitable hypothesis in-between the boundaries.

The problem with the most-general boundary is that early on in the learning process it might only contain a few literals, and therefore contain only very sparse information about the required spatial and structural characteristics of the tool and action. Using this as the working hypothesis will usually result in a very high proportion of negative examples, which is not desirable in the early steps of learning. Learning a tool use hypothesis which consists of, say, eight literals requires a great deal of experimentation since the general-to-specific learner must eliminate all hypotheses of length 1, then length 2, and so on.

Using the most-specific boundary is more promising, since it is more likely to produce positive examples and will also be able to learn longer clauses more quickly. However, inevitably the most-specific hypothesis contains a very large number of

literals early on in the learning process. When selecting a new example to test, the agent can usually only choose to satisfy a small subset of these literals, and the difficulty comes in choosing which literals to test. A most-general boundary helps to identify some of these literals and prevents the agent from overgeneralising.

The approach we take then, is to construct a working hypothesis by starting at the most-general boundary and working our way inwards. The exact details of how this done are detailed below, but it essentially involves starting with the most-general hypothesis as a "base" clause, and then adding literals from the most-specific clause to it. This allows us to select a hypothesis which is as specific as possible, but still remains within the most-general boundary.

In constructing a working hypothesis which most closely matches the most-specific boundary h_S we divide our approach in two. The tool pose hypothesis incorporates both structural literals (describing the physical composition, structure, and shape of the tool) and spatial literals (describing how the tool should be placed relative to other objects). We therefore first select a tool structure which best matches h_S, and then (assuming a tool with the chosen structure) try to select a tool pose which matches as closely as possible to that specified in h_S.

Tool selection

The basic idea of the tool selection algorithm is to find the object which is able to satisfy the most structural constraints in the most-specific hypothesis. Each object is scored for suitability according to the number of literals in h_S it is able to satisfy, and the one with the highest score is chosen. In addition to satisfying as many structural literals in h_S as possible, we require that the object satisfy **all** of the structural constraints in the most-general hypothesis h_G. If an object does not satisfy all of the structural literals in h_G it is assigned a score of zero.

The main steps of the algorithm are as follows. Using the most-specific and most-general clauses of the hypothesis $h \equiv$ `tool_pose(Tool,State)`:

1. For each potential tool object `Obj`, count the number of satisfiable tool structural literals in h_S
2. Assign a score of zero to any object which does not satisfy all of the struc-

tural literals in h_G.

3. Select the tool object with the highest score (greatest number of satisfied literals).

As an example of the tool selection algorithm, we once again consider the problem of choosing a suitable cup for collecting water from a tap. Figure 4.10 shows three candidate objects which the agent can choose from. Let us assume that the agent has the following very simple versions of the most-specific and most-general hypothesis (in practice the most-specific hypothesis would be much longer):

```
tool_pose_G(Cup,State) :- is_tap(Tap),
                          under(Cup,Tap,State),
                          has_handle(Cup,Handle).

tool_pose_S(Cup,State) :- is_tap(Tap),
                          under(Cup,Tap,State),
                          orientation(Cup,vertical),
                          shape(Cup,concave),
                          wider(Cup,Tap),
                          taller(Cup,Tap),
                          has_handle(Cup,Handle),
                          shape(Handle,rounded).
```

The algorithm requires that all literals in `tool_pose`$_G$`(Cup,State)` be satisfied — ignoring literals which relate to the spatial positioning of the tool, such as `under(Cup,Tap,State)`. Thus object `cup3` can be immediately eliminated because it does not have a handle. Of the remaining objects, `cup1` scores more highly than `cup2` because it satisfies more static literals in h_S. Note that the chosen object does not need to satisfy all the literals of h_S (in this case it fails to satisfy `shape(Handle,rounded)`).

Pose selection

In a similar way to the tool selection algorithm, the pose selection algorithm attempts to maximise the number of satisfied literals in the most-specific hypothesis

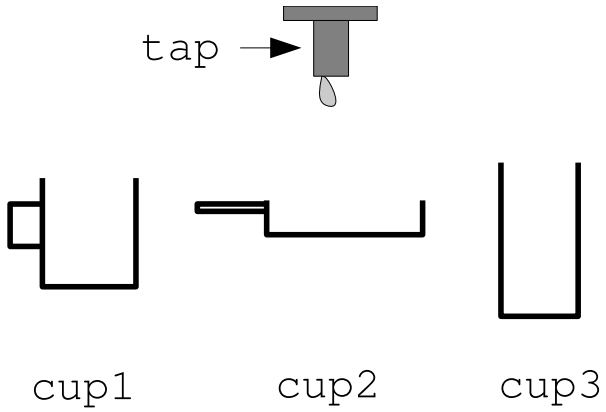


Figure 4.10: Selecting a tool which best matches the current most-specific hypothesis h_S. The tool which is able to be matched to the largest number of structural literals in h_S is chosen (see main text).

h_S. In this case however we are interested in satisfying the greatest number of spatial literals. This will allow the agent to place its selected tool in a pose which is as similar as possible to the previous positive examples it has observed.

The process we use for selecting a suitable subset of spatial literals follows a similar principle to the tool selection algorithm. We firstly demand that all spatial constraints in the most-general clause h_G must be satisfied — this ensures that the most important constraints are applied first. We then successively apply spatial constraint literals from the most-specific hypothesis h_S, checking at each step whether the cumulative constraints can be satisfied. This checking process is carried out by the agent's spatial constraint solver.

A more detailed description of the method is given in Algorithm 1. The algorithm is best illustrated with an example. Consider the agent testing a hypothesis in the cup problem. Assume that it has already selected the martini glass shown in Figure 4.11 as a suitable cup object to test. It then wishes to choose a spatial pose in which to place the tool. It does this by selecting a pose which satisfies as many spatial literals in h_S as possible.

Let us assume that the martini glass has parts `base`, `leftwall`, and `rightwall` as shown in the Figure. Further, let us suppose that with the martini glass substituted as the *Tool* parameter, the spatial literals in the agent's most-specific and most-general hypothesis are as follows:

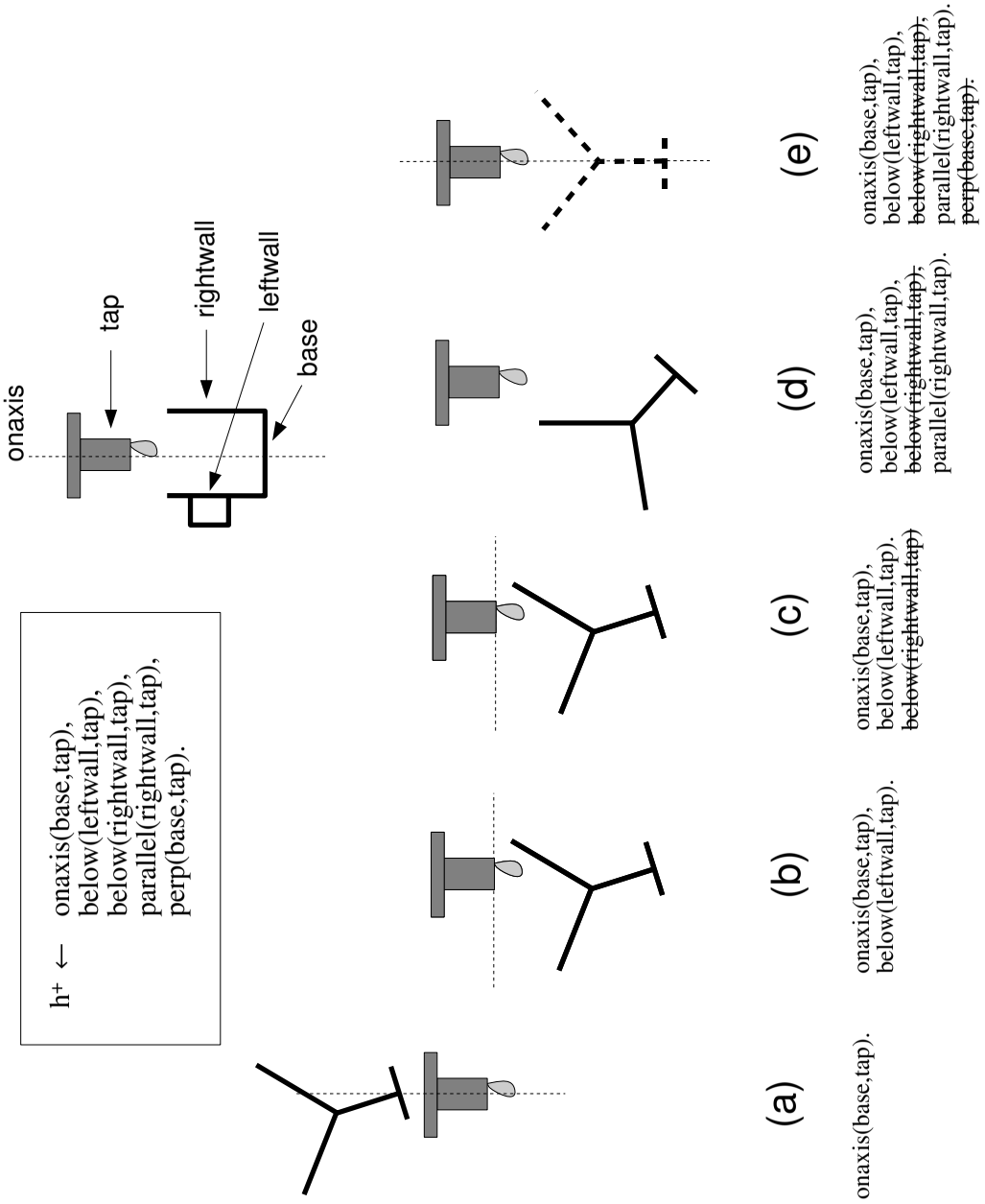


Figure 4.11: Illustration of the SELECT_POSE algorithm (Algorithm 1) for the cup problem. At the top of the figure is a definition of the current most-specific hypothesis h_S. Parts (a) through (e) show solutions to this hypothesis, with each applying a successively larger subset of the constraint literals. Literals which are found to be redundant (already satisfied at the previous step) or cannot be satisfied are excluded, as shown by strikethrough text. In step (c) a redundant literal `below(rightwall,tap)` is ignored, whilst step (e) shows a literal `perpendicular(base,tap)` which cannot be satisfied (given the other constraints). The solver therefore backtracks to (d), which turns out to be a negative example.

h_S :- below(rightwall,tap),
parallel(rightwall,tap),
below(leftwall,tap),
onaxis(base,tap)
perp(base,tap).

h_G :- onaxis(base,tap)

These spatial constraints say that the base of the martini glass should lie on the vertical axis of the tap, that both walls of the glass should lie below the tap, that the right wall should be parallel to the tap, and the base should be perpendicular. The mug shown at the top right of the Figure satisfies all these constraints, but as we shall it is not possible to satisfy the same set of constraints with the martini glass.

The complete set of constraints to be tested is constructed by starting with those in h_G, and then adding the remaining literals from h_S in random order.

Algorithm 1 Find a spatial pose satisfying the most-specific hypothesis.

SELECT_POSE

Let $\mathtt{A}_1, \mathtt{A}_2, \ldots, \mathtt{A}_n$ denote the spatial literals in the most-general clause h_G. Let $\mathtt{B}_1, \mathtt{B}_2, \ldots, \mathtt{B}_n$ denote a shuffled list of spatial literals from h_S, excluding any which are already represented in the $\mathtt{A}_i$. Finally, let C represent the current set of spatial constraints to be applied.

1: Add the most-general constraints to C (ie. let $\mathtt{C} \leftarrow \mathtt{A}_1, \mathtt{A}_2, \ldots, \mathtt{A}_n$)
2: Solve C and let P represent the tool pose solution
3: **for** each successive literal $\mathtt{B}_i$ in $\mathtt{B}_1, \ldots, \mathtt{B}_n$ **do**
4: If $\mathtt{B}_i$ is already true in P, discard $\mathtt{B}_i$ (a redundant constraint)
5: Else append $\mathtt{B}_i$ to the end of C, and find an updated pose solution P to the constraints in C
6: If no solution P exists, remove $\mathtt{B}_i$ from the end of C (incompatible constraint)
7: **end for**
8: Output the set of applied constraints C and the solution pose P which satisfies these constraints

Suppose this produces the following set of ordered constraints to test:

```
C :- onaxis(base,tap)
     below(leftwall,tap),
     below(rightwall,tap),
     parallel(rightwall,tap),
     perp(base,tap).
```

The agent starts at the top of this list of constraints, and adds the literals to the constraint solver one-by-one. At each step, the solver tries to find a solution pose for the tool which satisfies the constraints. If a valid pose is found the constraint literal is retained; otherwise the literal is discarded. Furthermore, if the new literal being added is already satisfied by the most recent solution pose then it is considered to be redundant and discarded.

The attempted solution at each of the five constraint-solving steps (corresponding to the five literals which are added) are illustrated in parts (a) to (e) in Figure 4.11. We comment on each step as follows:

(a) Shows a solution pose which satisfies just the first literal `onaxis(base,tap)`.

(b) The second constraint `below(leftwall,tap)` forces the glass below the tap.

(c) The third constraint `below(leftwall,tap)` is redundant because it is already satisfied using the pose found at the previous step. This constraint is therefore discarded.

(d) The fourth constraint `parallel(rightwall,tap)` can be satisfied by rotating the glass as shown in the Figure.

(e) The final constraint `perp(base,tap)` cannot be satisfied because it is incompatible with the earlier constraints (in particular, `parallel(rightwall, tap)`. This constraint is therefore discarded, and the final solution pose backtracks to the valid solution found in the previous step (which turns out to be a negative example).

The end result is that the solution pose shown in step (d) is returned as the pose to be tested. The agent will now carry out the tool action using this instantiation

of the hypothesis and discover that it is not a very sensible tool pose for collecting water. The state shown in (d) will then get added as a negative example of the tool-use task and the agent's hypothesis will be revised.

As we shall see in Section 4.3.5 this negative example will not affect the most-specific clause, which is generated purely through positive examples. However it will cause the most-general boundary to be generalised: that is, additional or different literals will be added to h_G. Since the spatial constraints in h_G are always given priority at the top of the list in the spatial constraint clause `C`, this will mean the agent will avoid repeating its current mistake.

The importance of the shuffling/randomisation step in the pose selection algorithm is worth emphasising. It ensures that the learner explores the space of spatial constraints around the most-specific hypothesis boundary. The order in which spatial constraints are applied is important (they are not commutative) and so by shuffling the most-specific literals we make it more likely that the learner will encounter positive examples, or at least a variety of negative examples.

4.3.5 Learning a new hypothesis: ILP algorithm

Given a set of labelled positive and negative examples of the target concept our ILP algorithm outputs two clauses representing most-specific and most-general boundaries on the hypothesis space.

The learning algorithm consists of two main steps:

1. Learning the most-specific boundary clause.
2. Learning the most-general boundary clause.

The most-specific boundary clause is built using only positive examples, via a process which seeks to find the minimal generalisation which covers a group of examples. We use a constrained form of least general generalisation (lgg) (Plotkin, 1970) as described in detail below. The most-general boundary clause is then built by direct generalisation of the most-specific boundary clause. Negative-based reduction is used to shorten the length of the most-specific clause without covering additional negative examples.

The resulting algorithm borrows heavily from the algorithm GOLEM (Muggleton and Feng, 1992). The primary difference is that GOLEM's bias is to learn the shortest, most-general hypothesis possible. In contrast, we maintain both a most-specific and most-general boundary on the hypothesis space.

The algorithm is re-run after each example is received, and the current hypothesis is rebuilt. Although the hypothesis is not incrementally varied, the time needed to re-learn the hypothesis after each example is on the order of a few seconds — much less than the time required to gather another example.

Initial hypothesis

If we denote the initial teacher's example as e_1 then the most-specific and most-general boundaries on the initial hypothesis space can be written:

$$h_S : -\texttt{saturation}(e_1)\texttt{.}$$
$$h_G : -\texttt{true}\texttt{.}$$

where $\texttt{saturation}(e_1)$ is the initial example saturated with the background knowledge (ie. the bottom clause). When performing this saturation we use the mode declarations described in Section 4.3.3 to generate the clause, and replace all real world objects with variables. The initial hypothesis boundaries are illustrated in Figure 4.12.

Most-specific boundary

The algorithm for learning the most-specific boundary is shown in Algorithm 2. At the high level this algorithm is identical to the search algorithm used in GOLEM (Muggleton and Feng, 1992). However, at the low-level we calculate lggs in a different way, as described below.

Our approach to computing the lgg of two examples is different to the usual definition (see Lavrac and Dzeroski (1994) for a description of the general approach to the lgg). We use the mode declarations to restrict the lgg, by demanding that the example clauses obey the mode requirements.

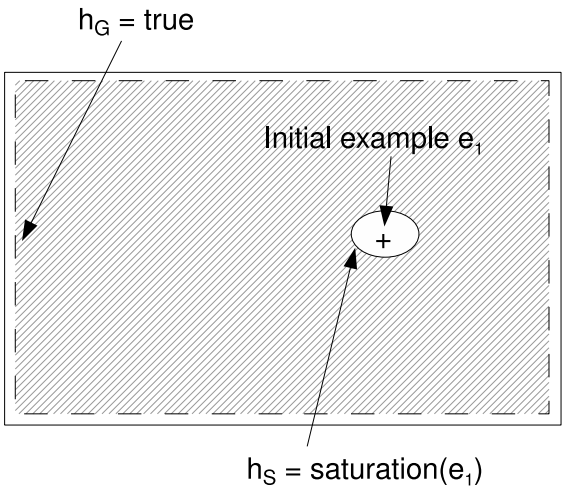


Figure 4.12: The initial hypothesis boundaries after observing the teacher's example. The most-specific boundary is given by the bottom clause of the example, whilst the most-general boundary clause is simply *true*.

As an example of the difference between our approach and the standard lgg defined in Plotkin (1970), consider the lgg of the following two clauses:

```
p(a,b) ← q(a), q(b).
p(c,d) ← q(c), q(d).
```

The conventional lgg of these two clauses would be:

```
p(A,B) ← q(A), q(B), q(C), q(D).
```

where the variables `A`, `B`, `C`, and `D` represent the inverse substitutions `a/c`, `b/d`, `a/d`, `b/c` respectively. In our work we only allow new variables to be introduced either in the head of the clause, or in output parameters in the body of the clause. Thus if `q(+letter)` and `p(+letter,+letter)` are mode declarations for the above clauses then the lgg would simply be:

```
p(A,B) ← q(A), q(B).
```

By restricting the lgg in this manner we are excluding some valid generalisations of the two clauses in order to constrain the hypothesis space. In effect we are demanding that the hypothesis clause has the same structural composition as the

Algorithm 2 Generate a most-specific boundary clause (from GOLEM).

MOST_SPECIFIC_HYP

Inputs:

- Pos, a set of positive examples
- Neg, a set of negative examples

1: $Pairs \leftarrow N$ random pairs of examples from Pos
2: **for all** pairs e_i, e_j in $Pairs$ **do**
3: Compute coverage of $rlgg(e_i, e_j)$
4: **end for**
5: $S \leftarrow$ pair e, e' with the best coverage
6: $Pos \leftarrow Pos -$ covered$(rlgg(S))$
7: **repeat**
8: $Es \leftarrow N$ random positive examples from Pos
9: **for all** e_i in Es **do**
10: $S' \leftarrow S \cup \{e_i\}$
11: $Score \leftarrow$ coverage of $rlgg(S')$
12: **end for**
13: $e_{best} \leftarrow$ example which gives best coverage $Score$
14: $S \leftarrow S \cup \{e_{best}\}$
15: $Pos \leftarrow Pos -$ covered$(rlgg(S))$
16: **until** Coverage of $rlgg(S)$ stops improving (or all examples covered)
17: **return** $rlgg(S)$, the most-specific boundary clause

example clauses, but only contains the common elements of both. The formal definition of our restricted lgg is given below.

It should be noted that our approach to calculating the lgg, although similar, is different to that used in GOLEM (Muggleton and Feng, 1992). In GOLEM mode declarations are used as a functional reduction step to delete literals from the lgg after it has been calculated. In our approach we use the mode declarations to apply restrictions to the lgg as it is being calculated. This results in an lgg which is in some cases more restricted, so that our hypothesis space is more constrained that the one searched by GOLEM.

Mode constrained lggs

We define our mode-restricted lgg as follows. Let a and b be constants appearing as the nth parameters of two body literals, and the mode of the parameter be indicated by $+$ or $-$. If the list of existing inverse substitutions generated by terms occurring earlier in the clause is $ExistingSubs$ then the mode-restricted lgg of these parameters is defined as:

1. $lgg(-a, -a) = a$
2. $lgg(+a, +a) = a$
3. $lgg(-a, -b) = \begin{cases} \text{X} & \text{if } X\!:\!a/b \text{ is a member of } ExistingSubs \\ \text{Y} & \text{otherwise, and } Y\!:\!a/b \text{ is added to } ExistingSubs \end{cases}$
4. $lgg(+a, +b) = \begin{cases} \text{X} & \text{if } X\!:\!a/b \text{ is a member of } ExistingSubs \\ \text{undefined} & \text{otherwise.} \end{cases}$

Parameters in the head of the clause are treated as output parameters for the purposes of calculating the lgg (ie. new variables are permitted in generalisations). The lgg of two literals is then defined in the usual way (Plotkin, 1970) as:

1. $lgg(p(a_1, a_2, \ldots, a_n), p(b_1, b_2, \ldots, b_n)) = p(V_1, V_2, \ldots, V_n)$
   where $V_n = lgg(a_n, b_n)$ and only if V_n is defined
2. $lgg(p(a_1, a_2, \ldots, a_n), p(b_1, b_2, \ldots, b_m)$ is undefined if $n \neq m$
3. $lgg(p(a_1, a_2, \ldots, a_n), q(b_1, b_2, \ldots, b_n)$ is undefined

We also define the lgg of two clauses C_1 and C_2 in the conventional manner: If $C_1 = \{A_1, A_2, \ldots, A_n\}$ and $C_2 = \{B_1, B_2, \ldots, B_n\}$ then

$$lgg(C_1, C_2) = \{\, L_{ij} \mid L_{ij} = lgg(A_i, B_j) \text{ is defined}\}$$

Most-general boundary

The most-general hypothesis boundary is derived directly from the most-specific boundary clause via negative-based reduction. Negative-based reduction was introduced in GOLEM as a method for generalising the lgg. Generalisation is achieved by removing literals from the lgg clause whilst attempting to preserve the clause's

coverage (ie. maintaining coverage of positive examples whilst avoiding covering negative examples).

Whereas calculating lggs involves learning only from positive examples, negative-based reduction focuses on using negative examples (as the name implies). It can be viewed as a method which expands the most-specific boundary outwards, stopping when it hits negative examples.

The negative-based reduction algorithm, from GOLEM, is as follows. Given a clause $A \leftarrow B_1, B_2, \ldots, B_n$ it involves the following steps:

1. Let B_i be the first literal in $B_1, B_2, \ldots, B_n$ such that $A \leftarrow B_1, \ldots, B_i$ covers no negative examples
2. If $i < n$ discard all remaining literals $B_{i+1}, \ldots, B_n$
3. Rotate the end literal to the front of the clause, and repeat the reduction process on the clause $A \leftarrow B_i, B_1, B_2, \ldots, B_{i-1}$
4. Repeat the rotation and reduction steps until the clause length remains unchanged during a full cycle (where each literal has been rotated to the front of the clause)

In our version of the negative-based reduction algorithm some care needs to be taken in evaluating coverage after each rotation step. In order for the resulting hypothesis to satisfy the mode-declarations it is necessary to "float" the front literal forward through the clause to the point at which its parameters are fully defined by those literals preceeding it.

Chapter 5

Experimental evaluation

The ultimate test of a tool-learning agent is whether – after some observation and practice – it is able to perform new versions of the task successfully. Given a new task and a selection of potential tool objects, an agent which has learnt a correct hypothesis should be able to select an appropriate tool and use it successfully to perform the task. In this chapter we present an evaluation of our tool-learning agent on three tool-use tasks.

5.1 Evaluation method and learning tasks

The work in this thesis falls into the category of *exploratory research* (Dietterich, 1990) in robot learning — existing approaches to tool and action learning are unable to solve the tool-learning problem presented here. Perhaps not surprisingly, there is as yet no set of standard learning problems for tool using agents. In the absence of a standard set of learning tasks we have chosen some interesting problems ourselves, being:

- A pull-tool task.
- A push-tool task.
- A ramp-tool task.

The results of these experiments are described in Sections 5.2 through 5.4.

In place of comparing our algorithm to a common baseline, we give a detailed trace and analysis of the learning algorithm on a typical experimental run of the

first learning task, and comment on the learnt hypotheses and average number of steps to learn. We follow this with a presentation and discussion of learning results on two other learning tasks, and conclude in Section 5.5 with a more general discussion of our approach, including its advantages and limitations.

5.2 Pull-tool problem: A detailed experimental trace and analysis

In this section we present a step-by-step trace of learning on the pull-tool problem, followed by a discussion of the learnt hypothesis.

5.2.1 The learning task

The objective of the pull-tool problem is to obtain a box which is placed out of reach in a closed-ended "tube" (see Figure 5.2). In order to solve this problem the robot must select an appropriate stick tool with a hook and use it to drag the box from the tube so that it can be picked up.

Initially the agent has no knowledge of "pulling" actions or that sticks with hooks can be used to bring out-of-reach objects within reach. To seed the learning process the agent is given a single task demonstration by a teacher agent. The agent can use this demonstration to explain how a tool can be used to obtain the box. It also provides an initial example of a suitable tool object, and of the relative spatial pose in which the tool must be placed.

The following actions are performed by the teacher during the demonstration:

1. Put the tool in the gripper.
2. Close the gripper.
3. Carry the tool and place it in a "pulling" pose.
4. Pull the box from the tube with the tool.
5. Place the tool aside.
6. Open the gripper.
7. Put the box in the gripper.

8. Close the gripper to grab the box.
9. Carry the box away.

Steps 3 and 4 correspond to components of the novel tool-use action which the agent must learn.

Following the teacher's demonstration a series of randomly generated versions of the pull-tool problem are presented to the learner. Each task features a different set of available tool objects, box and tube. The type of tool objects available in this task are illustrated in Figure 5.1; one of each type is available for each generated task. The learner is allowed to experiment at solving each task with the various tools which are available. Each time the agent succeeds at solving the task a new version of the problem is generated. The learning process is stopped once the agent has learnt a hypothesis equivalent to the target hypothesis (described below).

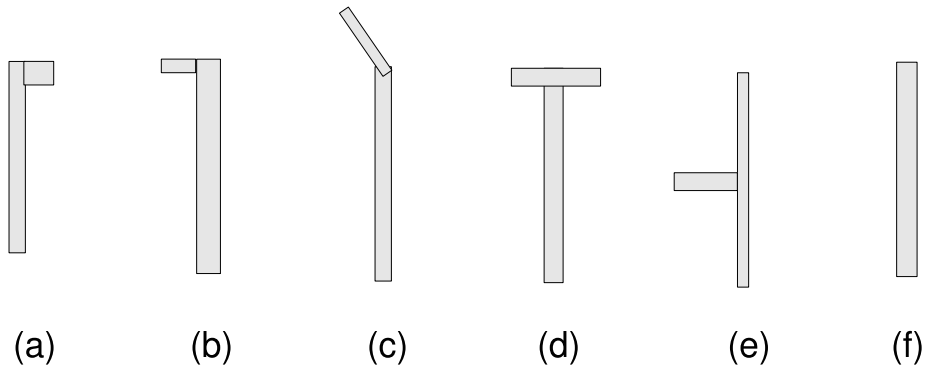


Figure 5.1: Types of tools available in the pull-tool problem. Each task has one of each type available. Depending on the location of the box in the tube, either (a) or (b) is the best tool for pulling.

Target hypothesis

A correct solution to this problem involves learning to select the correct type of tool object, placing it in the correct pulling pose, and then dragging the box from the tube. The learner will attempt to capture the required spatial and structural characteristics of the tool by learning a hypothesis describing the tool pose state (see Section 4.1 for a definition of the tool pose state).

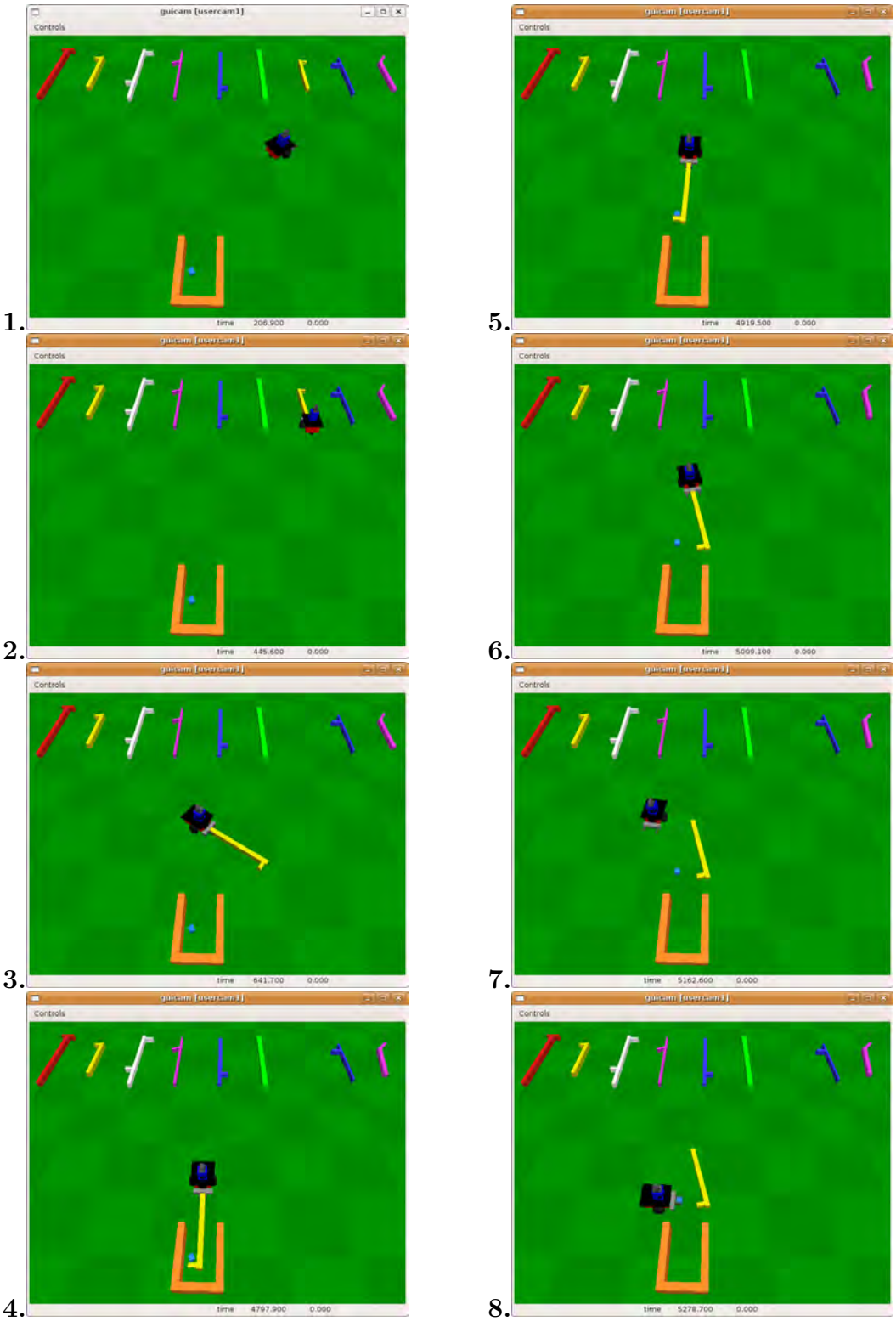


Figure 5.2: The pull-tool problem. A box is placed inside a "tube", out of reach of the robot. The robot must pick up an appropriate tool (steps 1 and 2), place it in a 'pulling' pose (steps 3 and 4), and then drag the box from the tube (step 5). The agent can then put the tool aside (step 6) and pick up the box (steps 7 and 8).

A positive example of the tool pose state for this problem is shown in Figure 5.3. Important spatial relationships which must be discovered include:

- the tool's hook must be placed behind the object, so it is touching
- the tool's handle should be (roughly) parallel to the walls of the tube
- the tool's handle should be touching the object on one side

Meanwhile, the required structural properties of a useful pulling tool include:

- it is the longest component of the object
- it has a single attachment (hook) which:
  - is attached to the front end of the tool
  - is attached at right angles
  - is attached on the same side of the tool as the box is in the tube (ie. if the box is on the left of the tube, a left-sided hook stick will be required)

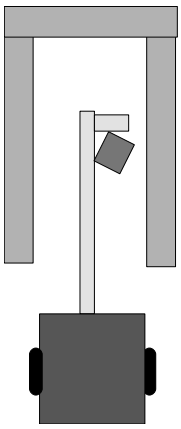


Figure 5.3: Illustration of the target hypothesis for the pull-tool problem. The correct solution involves having the tool parallel to the tube, with the box behind the hook and up against the side of the tool handle. Furthermore the hook should be near the end of the handle and on the appropriate side of the handle.

5.2.2 Background knowledge

Actions

The learner agent is provided with the following actions for picking up and moving objects (the corresponding action models are shown in Figure 5.4):

- `put_in_gripper(Object)`:
  Put an object in the gripper area
- `close_gripper(Object)`:
  Grip an object which is in the gripper area
- `move_obstacle(Object1,Object2)`:
  Move an obstacle (`Object1`) out of the way of another object (`Object2`)
- `open_gripper`:
  Open the gripper
- `remove_from_gripper(Object)`:
  Remove an object from the gripper area

In addition to these abstract actions, the agent also has two manipulation recognition models (Figure 5.5):

- `recognise_goto`
- `recognise_carry_obj(Obj)`

As discussed in Chapter 3, manipulation recognition models are used for identifying general actions whose sub-goals are unknown. The `recognise_goto` model allows recognition of agents moving to a new location, whilst the `recognise_carry_obj(Obj)` model is activated when another agent is observed moving with an object in its gripper.

State predicates

The abstract state of the world in the pull-tool problem is described by the state predicates shown in Figure 5.6. As in Chapter 4 we differentiate between *dynamic spatial* predicates, which have a state parameter argument and change value de-

grip(Obj)

PRE	¬gripping, in_gripper(Obj)
ADD	gripping
DEL	-
PRIMTV	closeGrip
MOVING	-

ungrip(Obj)

PRE	gripping, in_gripper(Obj)
ADD	-
DEL	gripping
PRIMTV	openGrip
MOVING	-

remove_from_gripper(Obj)

PRE	in_gripper(Obj), ¬gripping
ADD	empty_gripper
DEL	in_gripper(Obj)
PRIMTV	back
MOVING	robot

put_in_gripper(Obj)

PRE	forall(Tube:tube, ¬in(Obj,Tube)), empty_gripper, ¬gripping, forall(Obstacle:obj, ¬obstructing(Obstacle,Obj)
ADD	in_gripper(Obj)
DEL	empty_gripper
PRIMTV	fwd,back,rotleft,rotright
MOVING	robot

move_obstacle(ObjA,ObjB)

PRE	moveable_obj(ObjA) obstructing(ObjA,ObjB) in_gripper(ObjA), gripping
ADD	-
DEL	obstructing(ObjA,ObjB)
PRIMTV	fwd,back,rotleft,rotright
MOVING	robot, ObjA

Figure 5.4: Action models provided as background knowledge for the pull-tool problem.

recognise_goto

PRE	empty_gripper
MOVING	robot

recognise_carry_obj(Obj)

PRE	in_gripper(Obj), gripping
MOVING	robot, Obj

Figure 5.5: Manipulation recognition models provided to the agent for the pull-tool problem

pending on the spatial situation, and *static structural* predicates which have no state parameter (because the structure and shape of objects does not change).

Dynamic predicates:

- in_gripper(+obj,+state)
- touching(+obj,-obj,-side,+state)
- at_right_angles(+obj,+obj,+state)
- at_oblique_angle(+obj,+obj,+state)
- parallel(+obj,+obj,+state)
- onaxis(+obj,+obj,+state)
- onperpaxis(+obj,+obj,+state)
- in_tube(+obj,-tube,+state)
- in_tube_end(+obj,-tube,-end,+state)
- in_tube_side(+obj,-tube,-side,+state)
- obstructing(+obj,+obj,+state)

Static predicates:

- attached_side(+obj,-obj,-side)
- attached_end(+obj,-obj,-disttype)
- attached_angle(+obj,-obj,-angletype)
- num_attachments(+obj,-number)
- longest_component(+obj)
- attached_type(+obj,-obj,-attachtype)
- narrower(+obj,+obj)
- shorter(+obj,+obj)
- shape(+obj,-shape)
- closed_tube(-tube, -obj, -obj, -obj)

Figure 5.6: Mode declarations for state predicates used for describing the world in the pull-tube problem

5.2.3 Explanation-based learning of a new action model

The agent observes a demonstration of a teacher agent completing the task, as shown in Figure 5.2. The demonstration is supplied to the agent as a time sequence of primitive states $w_1, w_2, w_3, \ldots$ where each state w_n specifies the poses of all objects in the world. A primitive state snapshot w_n is taken every 0.1 seconds during the teacher's demonstration.

Motion-based segmentation

The learner agent begins by segmenting the demonstration trace using the motion-based segmentation process described in detail in Chapter 4.

The observed instantaneous speeds of the robot, tool and box objects during a typical demonstration are shown in Figure 5.7 (other objects remain stationary

during the demonstration and are not shown). For each of these objects, a hysteresis threshold is used to deterimine the times at which the object appears to be moving or stationary. The results of applying these movement thresholds to each object are shown in Figure 5.8.

These thresholded movements of each individual object are then merged to produce a single segmentation of the demonstration trace, resulting in the coloured graph at the bottom of Figure 5.8. Each coloured segment corresponds to a different combination of moving objects — for example, the red segments correspond to the teacher robot moving by itself, whilst the green segments correspond to the robot moving simultaneously with the tool object.

The segments identified in Figure 5.8 are listed in Table 5.1. At this point the agent has not yet identified specific actions executed by the teacher. In order to do so it must now try to match action models to each segment.

Segment	Moving objects	Teacher's action (unknown)
1	robot	Put tool in gripper
2	gripper	Close gripper
3	robot, tool	Put tool in pulling pose
4	robot, tool, box	Pull box with tool
5	robot, tool	Put tool aside
6	gripper	Open gripper
7	robot	Put box in gripper
8	gripper	Close gripper
9	robot, box	Carry away box

Table 5.1: Segmentation of the pull-tool problem.

Constructing an explanation of the teacher's demonstration

The agent can construct an explanation of the teacher's activities by matching abstract action models to the motion segments it has identified. An abstract model matches a segment if the action preconditions are true at the beginning of the segment and the effects have been achieved by the end of the segment. The moving objects in the segment must also match the moving objects named in the

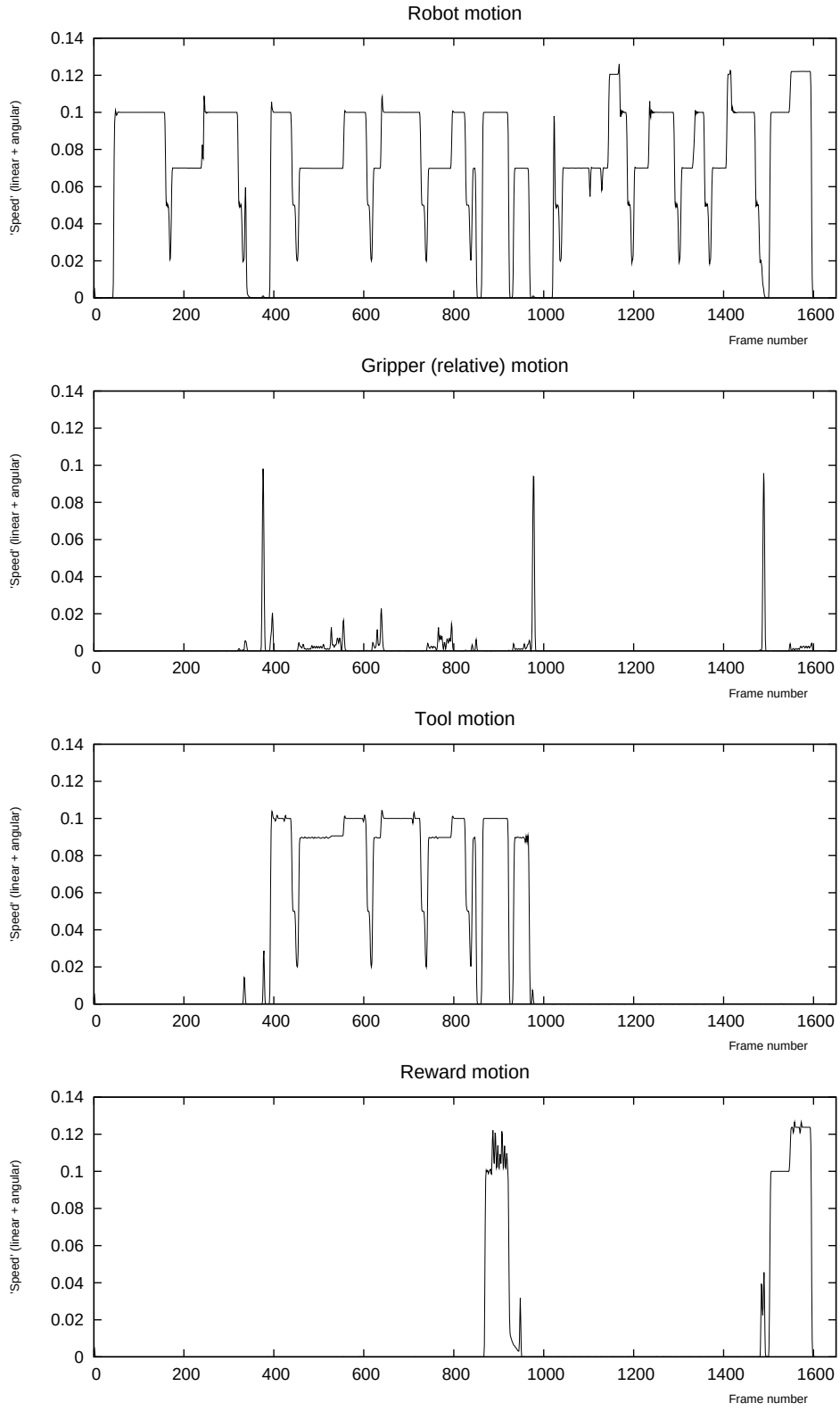


Figure 5.7: Observed speeds of the robot, gripper, tool, box during the teacher's demonstration.

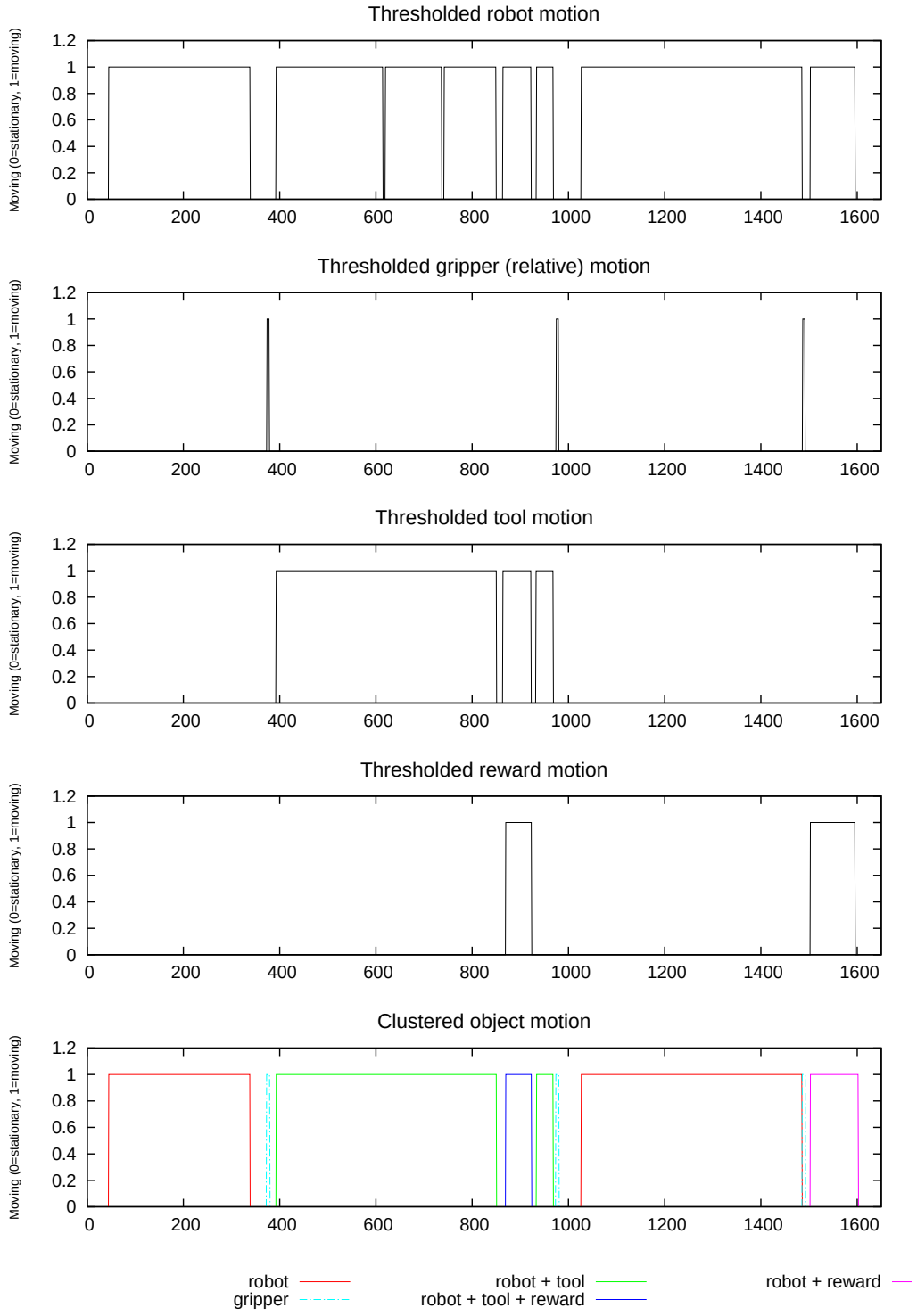


Figure 5.8: Robot and object motion during the teacher's demonstration, obtained from thresholding and clustering the observed speeds in Figure 5.7. The top three graphs show the individual motions of the robot, tool and box. The bottom graph shows the results of clustering these motions. Each colour represents a different combination of objects as indicated.

Seg	*Moving objects*	*Explanation*	*Match type*
1	robot	`put_in_gripper(hookstick)`	exact
2	gripper	`grip(hookstick)`	exact
3	robot, hookstick	`recognise_carry_obj(hookstick)`	partial
4	robot, hookstick, box	`??`	none
5	robot, hookstick	`move_obstacle(hookstick,box)`	exact
6	gripper	`ungrip(hookstick)`	exact
7	robot	`remove_from_gripper(hookstick)`, `put_in_gripper(box)`	exact
8	gripper	`grip(box)`	exact
9	robot, box	`recognise_carry_obj(box)`	partial

Figure 5.9: Explanations of each of the observed motion segments in the teacher's demonstration. Segment 4 represents a completely novel action since it does not match any of the learner's action models. Segments 3 and 9 represent partial matches, where the type of action (carrying or moving) is recognised but the exact sub-goal is not.

corresponding action model.

The result of applying this matching process to the motion segments in Figure 5.8 is shown in Figure 5.9. For each segment three possible types of matches are possible:

- exact: a single abstract action model (segments 1, 2, 5, 6, 8)
- exact: a sequence of abstract action models (segment 7)
- partial: a manipulation recognition model (segments 3 and 9)
- none: no match exists (segment 4)

Recall that manipulation recognition models, denoted by `recognise_*` (eg. `recognise_push` or `recognise_carry`), allow the agent to describe actions where the type of action is recognised (eg. pushing an object) but the specific goals of the action are not. For example, in segment 3 the learner recognised that the teacher is carrying the tool to a new location but it has no action model which describes the effects which occurred. Manipulation recognition models can be matched to these segments because they only require preconditions to be satisfied (the effects are empty). We describe segments which can be matched by manipulation recog-

nition models such as these as partially-matched. In tool actions these segments correspond to the tool "positioning" steps described in Chapter 4.

The most interesting segment in this trace is segment 4, which could not be matched to any known action model. Unmatched segments such as these represent unrecognised actions which involve the teacher interacting with objects in novel ways. In this case the teacher is pulling a box object with a tool, a type of action that does not exist in the learner's action descriptions.

New tool actions

The action model matching process allows the learner agent to identify novel action segments; the next step is to try and understand the purpose of these action segments. As described in Chapter 4, the learner can construct action models for these novel segments through a process similar to explanation-based learning (Mitchell et al., 1986). The construction of action models is based upon the idea that the sequence of observed actions must form a rational plan which achieves a goal.

Following this rationality assumption, the preconditions and effects of the missing (or incomplete) action models in the plan can be partially inferred by trying to build an explanation which contains no unsupported preconditions or unexplained effects. *Unsupported preconditions* are preconditions which are not enabled by the (known) effects of any preceeding actions (and which were not already true at the start of the demonstration). *Unexplained effects* are effects occurring in an action segment which are not explained by the the corresponding action model (if any), and which enable the preconditions of one or more actions occurring later in the demonstration.

Our explanation-based learning approach to action model construction involves three steps:

1. Identify any unsupported action preconditions
2. Identify any corresponding unexplained effects
3. Construct new action models which account for the unexplained effects and enable the unsupported preconditions in the explanation

In this case, two unsupported preconditions can be identified in the example demonstration trace:

- `obstructing(hookstick,box)`:
  a precondition of `move_obstacle(hookstick,box)`
- ¬`in_tube(box,tube)`:
  a precondition of `put_in_gripper(box)`

Each unsupported precondition can be matched with an unexplained effect occuring earlier in the demonstration. In this case, the literal `obstructing(hookstick, box)` occurs during segment 3 whilst the literal `in(box,tube)` is deleted during segment 4. Neither effect is accounted for by any of the action models in the original explanation.

The learner can form a consistent explanation by introducing new actions which account for these unexplained effects. Recall that the method used involves constructing new action models using the following principles:

- A new action model is constructed for any segment containing an unexplained effect which supports a later action precondition.
- The preconditions of the new action include the subset of literals which are true at the beginning of the segment and which are also the effects of preceeding actions in the demonstration.
- If the action segment has been matched to a manipulation recognition model, then the recognition model preconditions are included in the novel action preconditions.
- If two new actions occur in sequence, and the earlier action is a recognised positioning action (ie. has been matched to a manipulation recognition model) then add a `tool_pose` effect to the positioning action, and add the same literal as a precondition of the second action. The `tool_pose` literal represents the components of the tool pose state (Section 4.1) which were not learnt by explanation — such as literals describing the spatial positioning and structure of the tool. The definition of this derived predicate will be learnt by trial-and-error experimentation in Section 5.2.4.

The new action model created to explain segment 4, satisfying these principles, is as follows:

```
pull_from_tube( hookstick, box, tube)
PRE:   in_tube( box, tube),
       in_gripper( hookstick),
       gripping,
       tool_pose( hookstick, box)
ADD:   -
DEL:   in_tube(box,tube)
```

The following comments explain how the model was constructed (this construction step is automated in our system):

1. The parameters of the action are defined as the objects which are manipulated in the segment (`hookstick` and `box`), plus any other object mentioned in the action effects (`tube`).
2. The effects of this action are the unexplained effects in the segment — ie. ¬`in_tube(box,tube)`, which supports the `put_in_gripper(box)` action occurring in a later segment.
3. The `in_gripper( hookstick)` and `gripping` precondition literals arise because they are present in the effects of earlier actions, and are still true at the start of the segment.
4. The `tool_pose( hookstick,box)` literal has been added to the precondition, because the action is preceeded by a novel positioning step (it was matched with the `recognise_carry_obj(hookstick)` recognition model). As we shall see below, the same literal will be added as an effect of the positioning step action model, so that the `pull_from_tube( hookstick, box, tube)` action can only be enabled by this positioning step.

Since the positioning segment (segment 3) was matched by a manipulation recognition model (`recognise_carry_obj(hookstick)`), we use this as the basis for building the positioning action. The preconditions of the action and allowed motion primitives are taken from the corresponding recognition model. As described above, the primary effect of the action is defined to be to achieve the

tool_pose predicate which enables the tool use action in segment 4. If we assign the action the name position_tool then the action model generated is:

```
position_tool( hookstick, box)
PRE:   in_gripper( hookstick),
       gripping
ADD:   tool_pose( hookstick, box),
       obstructing( hookstick, box)
DEL:   -
```

Note that a second effect of this action is to achieve the obstructing(hookstick,box) precondition of move_obstacle(hookstick,box) (segment 5). This effect can be regarded as a side-effect of putting the tool in position — an effect which needs to be undone by the move_obstacle action before the box is picked up.

The pull_from_tube and position_tool actions defined above allow a complete and consistent explanation of the teacher's actions to be constructed. In order to apply them to new situations involving different objects, the object parameters are generalised to variables, so that the final action models are shown below. These action models were generated automatically by our agent, following the construction rules described above.

```
position_tool( Tool, Box)
PRE:   in_gripper( Tool),
       gripping
ADD:   tool_pose( Tool, Box),
       obstructing( Tool, Box)
DEL:   -
pull_from_tube(Tool,Box,Tube)
PRE:   tool_pose( Tool, Box),
       in_gripper( Tool),
       gripping,
       in_tube( Box, Tube)
ADD:   -
DEL:   in_tube(Box, Tube)
```

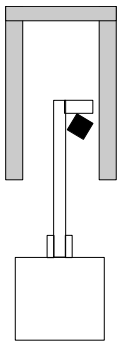


Figure 5.10: The tool pose state for the teacher's demonstration of the task (a positive example).

5.2.4 Learning from experimentation

Having constructed the initial action model to represent the novel tool action to be learnt, the agent must experiment and refine this model through trial and error. The agent can do this by learning a definition of the `tool_pose` predicate which appears in the precondition of the action. This derived predicate represents the components of the tool pose state (see Section 4.1) which were not able to be learnt through explanation of the teacher's example. This includes any spatial or structural preconditions which describe the correct positioning, shape or structure of the tool.

The teacher's example and initial hypothesis

As described in Chapter 4, the learner is able to extract an initial positive example of the tool pose state from the teacher's demonstration trace. This initial example is illustrated in Figure 5.10. If we label this state as `s1` then the initial most-general (h_G) and most-specific (h_S) boundaries on the agent's hypothesis can be written:

$$\texttt{tool\_pose}_G\texttt{(Tool,Box,State)} \texttt{ :- true.}$$
$$\texttt{tool\_pose}_S\texttt{(Tool,Box,State)} \texttt{ :- saturation(s}_1\texttt{).}$$

where `saturation(`$\texttt{s}_1$`)` is short-hand for the lengthy list of literals in the bottom clause describing the state `s1`.

The full bottom clause is reproduced in Figure 5.11 for illustrative purposes — note that it is a very specific and very lengthy clause. In the figure we have split the literals into static and dynamic components. The dynamic literals represent a description of the *spatial* relationships which exist between objects in the initial example state. The static literals represent a *structural* description of the tool and other objects — that is, how the objects are composed, their dimensions and shape.

Clearly the initial most-general boundary clause is too general (it is simply `true`), and the most-specific boundary too specific. The learner's task then is to refine these hypothesis boundaries to provide a more accurate and useful description of correct tool use.

Experimentation by the learner agent

The learner is given a series of new learning tasks, each featuring a new tube, box, and a different selection of tools. At each step the learner selects a suitable tool and attempts to use it to obtain the box from the tube. If the attempt is successful a positive example of the tool pose state is recorded, and a new task is generated. If the attempt fails, the learner records a negative example and the learning task is reset (whereupon the agent either chooses a new tool, or tries to use the same one in a different manner).

In the remainder of this section we present a trace of the examples collected by the learner during this experiment. Fifteen examples, including the teacher's example, were needed to learn a successful hypothesis (the experiment terminates when the agent succeeds at solving five successive versions of the task). The first 12 example states are illustrated in Figure 5.12 and are labelled `s1` to `s12`, where `s1` is the first positive example provided by the teacher.

Tool selection

The first task encountered by the learner is illustrated in Figure 5.13 with available tool objects `leftstick2`, `rightstick2`, `anglestick2`, `tstick2`, `middlestick2`, and `simplestick2`. The "2" suffix on each object indicates it is from the second

```
tool_pose_S( Tool, Box, State):-

% static literals:

attached( Tool, Hook),                                  narrower( Tool, TubeLeft),
num_attachments( Tool, 1),                              narrower( TubeLeft, Box),
num_attachments( Box, 0),                               narrower( Hook, TubeLeft),
longest_component( Tool),                               narrower( Tool, TubeRight),
narrower( Tool, Box),                                   narrower( TubeRight, Box),
shorter( Box, Tool),                                    narrower( Hook, TubeRight),
shape( Tool, sticklike),                                narrower( Tool, TubeBack),
shape( Box, boxlike),                                   narrower( TubeBack, Box),
closed_tube( Tube, TubeLeft, TubeRight, TubeBack),      narrower( Hook, TubeBack),
attached_side( Tool, Hook, right),                      narrower( TubeBack, TubeLeft),
attached_side( TubeLeft, TubeBack, right),              narrower( TubeBack, TubeRight),
attached_side( TubeRight, TubeBack, left),              shorter( Hook, Tool),
attached_end( Tool, Hook, front),                       shorter( Tool, TubeLeft),
attached_end( TubeLeft, TubeBack, front),               shorter( Tool, TubeRight),
attached_end( TubeRight, TubeBack, front),              shorter( TubeBack, Tool),
attached_angle( Tool, Hook, right_angle),               shorter( Box, Hook),
attached_angle( TubeLeft, TubeBack, right_angle),       shorter( Box, TubeLeft),
attached_angle( TubeRight, TubeBack, right_angle),      shorter( Hook, TubeLeft),
attached_type( Tool, Hook, end_to_end),                 shorter( TubeBack, TubeLeft),
attached_type( TubeLeft, TubeBack, end_to_end),         shorter( Box, TubeRight),
attached_type( TubeRight, TubeBack, end_to_end),        shorter( Hook, TubeRight),
num_attachments( Hook, 0),                              shorter( TubeBack, TubeRight),
num_attachments( TubeLeft, 1),                          shorter( TubeBack, TubeLeft),
num_attachments( TubeRight, 1),                         shorter( Box, TubeBack),
num_attachments( TubeBack, 0),                          shorter( Hook, TubeBack),
narrower( Hook, Tool),                                  shape( TubeLeft, sticklike),
narrower( Hook, Box),                                   shape( TubeRight, sticklike),
                                                        shape( TubeBack, sticklike),

% dynamic literals:

in_gripper( Tool, State),                       at_oblique_angle( Box, Hook, State),
touching( Tool, Box, right, State),             at_oblique_angle( Box, TubeLeft, State),
at_oblique_angle( Tool, Box, State),            at_oblique_angle( Box, TubeRight, State),
in_tube( Box, Tube, State),                     at_oblique_angle( Box, TubeBack, State),
in_tube_end( Box, Tube, front, State),          parallel( Tool, TubeLeft, State),
in_tube_side( Box, Tube, right, State),         parallel( Tool, TubeRight, State),
touching( Hook, Box, back, State),              parallel( Hook, TubeBack, State),
at_right_angles( Hook, TubeLeft, State),        in_tube( Hook, Tube, State),
at_right_angles( Hook, TubeRight, State),       in_tube_end( Hook, Tube, front, State),
at_right_angles( Tool, TubeBack, State),        in_tube_side( Hook, Tube, right, State).
```

Figure 5.11: The initial most-specific hypothesis clause h_S, split into static and dynamic components. The dynamic component consists of spatial literals with a `State` parameter. These spatial literals are the constraints which can be passed to the constraint solver. The static component consists of literals which do not change (they describe the structure and shape of objects), and they therefore have no `State` parameter.

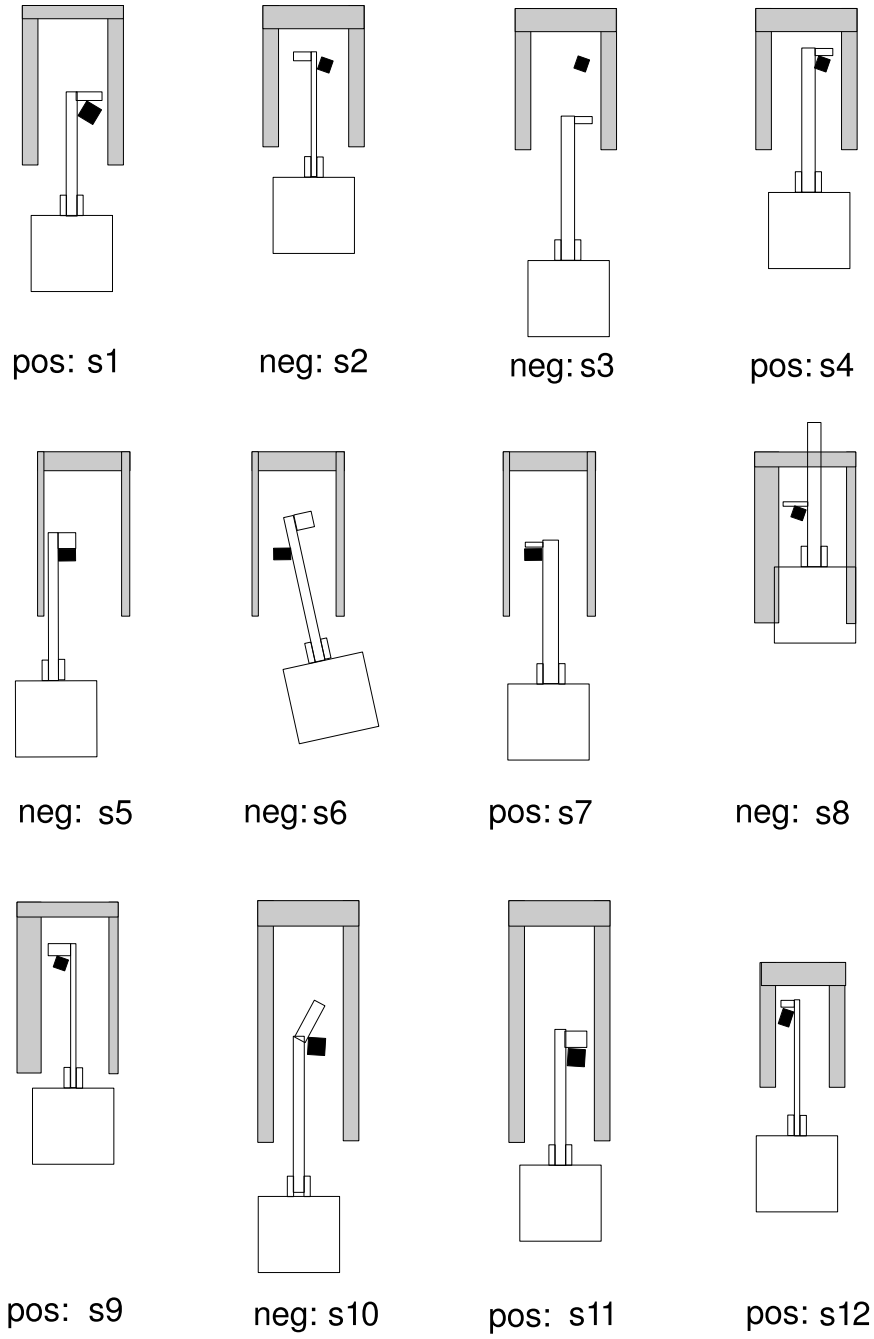


Figure 5.12: Tool pose state examples generated during learning by experimentation.

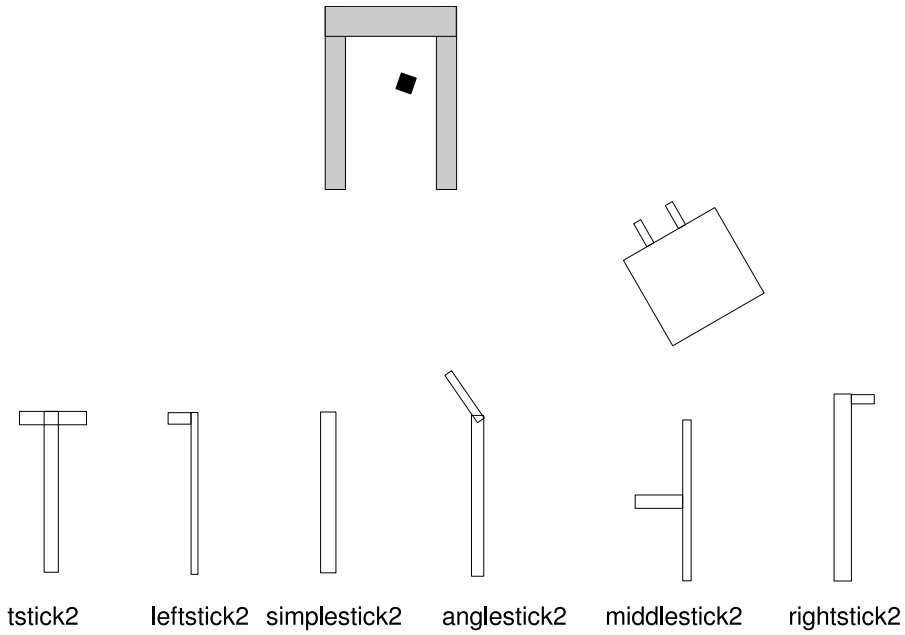


Figure 5.13: The first task encountered by the agent after the teacher's demonstration.

version of the task (the teacher's task being the first), and ensures that each object in the series of tasks is named uniquely.

The agent selects the best potential tool by a process of similarity-matching described in Chapter 4. A tool is scored according to the number of structural (static) literals it can satisfy in the most-specific clause, and *assigned a score of zero if it fails to satisfy the literals in the most-general clause*. The literals in the most-general clause represent "essential" properties of the tool.

Since the most-general hypothesis initially contains no structural constraints no tools can be immediately excluded. Counting structural literals in the most-specific hypothesis gives the following ordering from best to worst match:

- `leftstick2`
- `rightstick2`
- `middlestick2`
- `anglestick2`
- `tstick2`

- `simplestick2`

Both `leftstick2` and `rightstick2` have scored highly because they contain the most structural similarities to the original example — namely a right-angled "hook" attached to the end of stick. The reason that the left-sided stick scores higher is because it satisfies some additional literals that the right-sided stick does not (eg. `narrower(Tool,TubeLeft)`). These additional literals are irrelevant to the task, but the agent is unable to distinguish between relevant and irrelevant literals. The agent therefore selects `leftstick2` to test.

Pose selection

The next step is for the agent to generate a spatial pose which satisfies the current hypothesis. As detailed in Chapter 4 this is achieved by starting with the spatial constraints in the most-general clause, and incrementally adding as many spatial constraints from the most-specific clause as possible. At each step the agent uses its spatial constraint solver to check whether a valid solution pose exists.

As in the tool selection step, the most-general hypothesis clause imposes no constraints. The most-specific clause, on the other hand contains a list of potential spatial constraints (see Figure 5.11). The agent substitutes the relevant ground objects (eg. `leftstick2`) into these literals, and then shuffles them randomly to give the following list of ground spatial constraints to be applied to the tool:

```
parallel(leftstick2, tube_leftwall2, State),
at_oblique_angle(leftstick2, box2, State),
in_tube(leftstick_hook2, tube2, State),
touching(leftstick2, box2, right, State),
in_tube_end(leftstick_hook2, tube2, front, State),
at_right_angles(leftstick_hook2, tube_rightwall2, State),
at_right_angles(leftstick2, tube_backwall2, State),
at_oblique_angle(box2, leftstick_hook2, State),
parallel(leftstick2, tube_rightwall2, State),
at_right_angles(leftstick_hook2, tube_leftwall2, State),
touching(leftstick_hook2, box2, back, State),
```

```
parallel(leftstick_hook2, tube_backwall2, State),
in_tube_side(leftstick_hook2, tube2, right, State)
```

This list of literals is passed to the constraint solver which attempts to apply the shuffled constraints one-by-one — any constraint which cannot be satisfied is ignored. (The one exception to this rule are constraints containing ground spatial constants such as `front, back, left` and `right`. In this case the solver tries to relax the constraint by allowing the ground constant to be variable.)

The order in which spatial constraints are applied is important, because they often conflict. Thus constraints appearing (randomly) near the top of the shuffled list are more likely to be preserved, whilst constraints near the bottom are often ignored. In the current example, the solver finds a solution pose which preserves the following spatial constraints (the constraints which could not be satisfied are shown with strikethrough text):

```
parallel(leftstick2, tube_leftwall2, State),
at_oblique_angle(leftstick2, box2, State),
in_tube(leftstick_hook2, tube2, State),
touching(leftstick2, box2, right, State),
~~in_tube_end(leftstick_hook2, tube2, front, State),~~
at_right_angles(leftstick_hook2, tube_rightwall2, State),
at_right_angles(leftstick2, tube_backwall2, State),
at_oblique_angle(box2, leftstick_hook2, State),
parallel(leftstick2, tube_rightwall2, State),
at_right_angles(leftstick_hook2, tube_leftwall2, State),
~~touching(leftstick_hook2, box2, back, State),~~
parallel(leftstick_hook2, tube_backwall2, State),
in_tube_side(leftstick_hook2, tube2, right, State)
```

The ground pose satisfying these constraints is shown as state $s2$ in Figure 5.12. Of particular note is the fact that the agent was unable to satisfy the condition `touching(leftstick_hook2, box2, back, State)`, which specifies that the back side of the hook must be touching the box. This is obviously because the hook is on the wrong side of the handle!

Having generated this ground pose for the tool, the agent attempts to execute the action. This involves placing the tool in the desired pose, and then attempting to move the tool and box out of the tube. This fails of course and the agent records the tool pose state $s2$ as a negative example (see Figure 5.12).

Refining the hypothesis

Now that the agent has collected a second example it is able to refine its hypothesis. Recall from Chapter 4 that negative examples constrain the most-general boundary of the hypothesis space. Therefore when a negative example is received the learner re-runs its algorithm for learning the most-general clause. This is done by negative-based reduction of the existing most-specific clause.

In this case the resulting most-general boundary clause h_G is:

```
tool_pose_G( Tool, Box, State) :-
        attached_side( Tool, Hook, right).
```

This constraint says that the tool must have an attached body (`Hook`) on the right-hand side. This specialisation of the most-general clause means that the hypothesis no longer covers the negative example $s2$.

Note that the most-specific boundary remains unchanged, since it is generated purely from the lgg of positive examples. Until more positive examples are found the most-specific clause will remain as it is.

Generating a new example (s3)

Since the agent has yet to solve the given task successfully the problem remains the same, but the tools and objects are reset to their starting positions[1]. Once again the agent must select a tool and a spatial pose to test which is consistent with its (revised) hypothesis. This follows the same process discussed above for example $s2$ except now the most-general hypothesis boundary comes into play.

Since the most-general clause now contains the restriction `attached_side( Tool, Hook, right)` this effectively eliminates `leftstick2` from consideration. The structural literals in the most-general clause must be satisfied when selecting

[1]This is primarily a time-saving device.

a tool — any tool which fails to satisfy a most-general literal is assigned an overall similarity score of zero. The agent therefore selects `rightstick2` as the best tool.

The ground tool pose is now generated with the newly selected tool. The most-general clause again contains no "compulsory" spatial constraints, whilst the most-specific clause has the same set of spatial constraints used above in the previous example (except with a different tool parameter). These constraint literals are shuffled randomly before the constraint solver attempts to apply them one at a time. This results in the following constraints being applied (once again, unsatisfiable literals are shown with strikethrough text):

```
at_right_angles(rightstick2, tube_backwall2, State),
at_oblique_angle(rightstick2, box2, State),
in_tube_end(rightstick_hook2, tube2, front, State),
~~touching(rightstick_hook2, box2, back, State),~~
at_right_angles(rightstick_hook2, tube_leftwall2, State),
parallel(rightstick_hook2, tube_backwall2, State),
in_tube(rightstick_hook2, tube2, State),
parallel(rightstick2, tube_leftwall2, State),
~~touching(rightstick2, box2, right, State),~~
in_tube_side(rightstick_hook2, tube2, right, State)
at_right_angles(rightstick_hook2, tube_rightwall2, State),
at_oblique_angle(box2, rightstick_hook2, State),
parallel(rightstick2, tube_rightwall2, State)
```

The corresponding ground pose for the tool is shown as state $s3$ in Figure 5.12. The reason the tool has been placed in a rather odd position is due to the ordering of the shuffled spatial constraints. In this case the literal `in_tube_end(rightstick_hook2, tube2, front, State)` was applied near the top of the list, which meant that the tool hook was constrained to lie in the front half of the tube. As a consequence neither of the `touching` predicates could be satisfied[2].

[2]It could be argued that certain predicates (eg. `touching`) should always be given higher precedence in the constraint satisfaction process to avoid situations like this. However, in some tool-use scenarios it is not desirable for the tool to actually touch the object it is interacting with — pouring water from a jug into a cup for example. Thus these sorts of precendence-based heuristics may be unlikely to be significant in general.

Clearly state $s3$ is not a good tool pose state. The agent soon discovers this when it tries to use the tool from this position. The action fails and the agent records $s3$ as a negative example.

Hypothesis revision

Having received a negative example the agent revises its most-general hypothesis h_G once again, via negative-based reduction of the most-specific hypothesis. It becomes:

```
tool_pose_G( Tool, Box, State) :-
        attached_side( Tool, Hook, right),
        touching( Hook, Box, back, State).
```

The new clause requires that the back of the hook be touching the box. The most-specific hypothesis remains unchanged since there have been no additional positive examples.

A positive example (s4)

The tool and pose selection steps are repeated in a similar manner to collect another example state: $s4$. In this case the agent selects the same `rightstick2` tool it was using in $s3$, since it still satisfies the most-general clause. The spatial constraints are again extracted from the most-specific clause and shuffled before being applied. However, this time the most-general clause also contains a spatial constraint `touching( Hook, Box, back, State)`. This spatial constraint is applied first, before any of the literals from the most-specific clause. This ensures that the ground pose satisfies the most-general clause.

The resulting tool pose is shown as state $s4$ in Figure 5.12. In this case the spatial constraints have been applied in an order which produces a "correct" pose. The agent places the tool in this pose and pulls the box from the tube successfully — a positive example can therefore be recorded.

Revision of the most-specific clause

Since the learner has just collected a positive example, it is able to revise its most-specific clause. It does this by computing the lgg of the saturated clauses representing the two positive examples it has observed ($s1$ and $s4$), a process described in detail in Chapter 4.

In general the lgg produces a shortened most-specific clause, since literals which are not present in both examples are eliminated. For example, the literal `narrower(Tool,Box)` which was present in the original most-specific clause is dropped because in $s4$ the tool is wider than the box.

Literals with matching predicate names but containing differing ground constants may be generalised by the lgg process. For example, the literal

```
in_tube_end( Tool, Tube, front, State)
```

from $s1$ and

```
in_tube_end( Tool, Tube, back, State)
```

from $s4$ are generalised in the lgg to give:

```
in_tube_end( Tool, Tube, X, State)
```

where `X` indicates a new variable which can be satisfied by either `front` or `back`. This generalisation allows the hook to be placed anywhere in the tube (front or back) and avoids repetitions of the situation in $s3$ where the tool was forced into the front half of the tube (producing a negative example).

We will not reproduce the full lgg here since it is still quite lengthy (see Figure 5.14 for another full listing of the lgg). However the new spatial literals involving the tool are as follows:

```
parallel( Tool, TubeLeft, State),
at_oblique_angle( Tool, Box, State),
in_tube( Hook, Tube, State),
touching( Tool, Box, right, State),
in_tube_end( Hook, Tube, X, State),
```

```
at_right_angles( Hook, TubeRight, State),
at_right_angles( Tool, TubeBack, State),
at_oblique_angle( Box, Hook, State),
parallel( Tool, TubeRight, State),
at_right_angles( Hook, TubeLeft, State),
touching( Hook, Box, back, State),
parallel( Hook, TubeBack, State),
in_tube_side( Hook, Tube, right, State)
```

Thus the only spatial generalisation which occurred in this step was the generalisation of `in_tube_end( Hook, Tube, front, State)` as described above.

The most-general clause h_G remains unchanged.

Further examples

A new task is generated for the agent to tackle, since the previous task has been solved successfully. The cycle of generating examples and updating the learner's hypothesis now continues. Since the process has already been illustrated in detail we will step more quickly through the remaining examples, pointing out the interesting features of the algorithm along the way.

- **Example s5 (neg)**: The agent selects the right-hook tool because it is most similar to the two positive examples seen previously. It also manages to place the tool in a pose very similar to the previous positive examples. Unfortunately this produces a negative example because the desired pose is unreachable. The most-general hypothesis is revised to:

  ```
  tool_pose_G( Tool, Box, State) :-
          attached_side( Tool, Hook, right),
          in_tube_side( Hook, Tube, right, State),
          at_oblique_angle( Tool, Box, State).
  ```

- **Example s6 (neg)**: The agent persists with the right-side hook tool, and tries to place it in a pose satisfying h_G and as many literals from h_S as

possible. This results in the pose shown in state $s6$ in Figure 5.12, which is another negative example. Note how in $s6$ the tool is touching the box on the left hand side. This results from the constraint solver trying to apply the constraint `touching(Tool,Box,right,State)`, but failing to find a valid solution — so it instead tries to impose the more general condition `touching(Tool,Box,_,State)` (where `_` indicates a variable which can be bound as required). A solution pose is then found which allows the tool to be touching the box.

After revising its hypothesis the new most-general clause is:

```
tool_pose_G( Tool, Box, State) :-
        attached( Tool, Hook),
        narrower( Hook, Box),
        touching( Hook, Box, back, State).
```

- **Example s7 (pos)**: The new h_G forces the agent to select a different tool — one with a hook narrower than the box. This requirement is consistent with the observed positive examples, but is not actually useful. Most of the tools satisfy this constraint (excluding the tool it was just using), and the highest scoring tool is again the left-sided hook stick.

  This time there is only a single spatial constraint to satisfy in the most-general clause. This gives the agent some flexibility in attempting to apply the remaining spatial constraints from the most-specific clause h_S. After applying `touching(Hook,Box,back,State)` the constraint solver is able to satisfy most of the remaining constraints in h_S. As in the previous example, it was unable to satisfy `touching(Tool,Box,right,State)` and instead relaxed this constraint to `touching(Tool,Box,_,State)`.

  The resulting pose is a successful one and the agent records a positive example. The most-specific clause h_S can now be generalised by computing the lgg of the positive examples. This produces an important generalisation which we now explain. The previous most-specific clause contained the literals

```
attached_side( Tool, Hook, right),
in_tube_side( Box, Tube, right, State)
```

whereas the new example contains the corresponding terms

```
attached_side( Tool, Hook, left),
in_tube_side( Box, Tube, left, State)
```

The lgg of these two terms produces the generalisation:

```
attached_side( Tool, Hook, Side),
in_tube_side( Box, Tube, Side, State)
```

where `Side` is a new variable. The important thing about this generalisation is that it links the side of the tube on which the box appears to the side of the tool on which the hook should appear. Thus if the box is on the right-hand side of the tube, the new h_S suggests that it is desirable for the hook to also be on the right-hand side. A tool will now score more highly in the tool-selection step if it matches this constraint.

Another interesting feature of the new h_S is that more of the spatial constraints have dropped out: specifically those requiring the tool and box to be at an oblique angle. The remaining spatial constraints in h_S are:

```
touching( Tool, Box, Side, State),
touching( Hook, Box, back, State),
in_tube( Hook, Tube, State),
in_tube_end( Hook, Tube, _, State),
in_tube_side( Hook, Tube, Side, State),
at_right_angles( Hook, TubeRight, State),
at_right_angles( Tool, TubeBack, State),
at_right_angles( Hook, TubeLeft, State),
parallel( Hook, TubeBack, State),
parallel( Tool, TubeRight, State),
parallel( Tool, TubeLeft, State)
```

Many of these literals are redundant, but importantly the order in which they are applied is unimportant. The "problematic" literals have already been weeded out and these remaining constraints are commutative (at least in the tube problem). Redundant spatial literals can also be detected by the constraint solver as we discuss following example $s12$.

- **Example s8 (neg)**: A new learning task is generated. The most-general hypothesis h_G is unchanged from example $s6$ so the agent selects a tool with a narrow hook, the best of which is a `middlestick` as shown in Figure 5.12. This time the agent is unable to find a valid solution pose which satisfies h_G, since the `touching( Hook, Box, back, State)` causes a collision in each position. The learner therefore adds the selected pose as a negative example of the hypothesis.

  The most-general hypothesis becomes:

  ```
  tool_pose_G( Tool, Box, State) :-
          in_tube_side( Box, Tube, Side, State),
          attached_side( Tool, Hook, Side),
          narrower( Hook, Box),
          attached_end( Tool, Hook, back).
  ```

- **Examples s9 to s12**: No tool satisfying the new structural constraints of h_G exists (ie. a tool with a left-sided, narrow hook, attached at the end). The learner therefore backtracks and generates an alternative most-general clause:

  ```
  tool_pose_G( Tool, Box, State) :-
          in_tube_side( Box, Tube, Side, State),
          attached_side( Tool, Hook, Side),
          touching( Hook, Box, back, State),
          attached_end( Tool, Hook, back).
  ```

  This allows the agent to choose the left-sided hook stick, and achieve the successful example shown as $s9$ in the figure. The hypothesis is once again

revised, and a new learning task generated. In the remaining examples $s10$ through $s12$ only one further negative example is encountered ($s10$). This negative example forces h_G to include the restriction `attached_angle( Tool, Hook, rightangle)`. After each additional positive example the most-specific hypothesis h_S based on the lgg is revised. The learning experiment is terminated after the agent successfully solves three more tasks (not shown in the Figure), meaning that it has solved five consecutive tasks without failure ($s11$ through $s15$). The final form of the hypothesis is given below.

The learnt hypothesis

The final form of the most-general clause h_G is:

```
tool_pose_G( Tool, Box, State) :-
        in_tube_side( Box, Tube, Side, State),
        attached_side( Tool, Hook, Side),
        touching( Hook, Box, back, State),
        attached_angle( Tool, Hook, rightangle),
        attached_end( Tool, Hook, back).
```

This states that the tool should have a hook attached on the same side as the box is in the tube, and the hook should be at a right-angle at the end of the handle. It also contains a spatial literal constraining the back of the hook to be touching the box. Note that this clause does not, by itself, contain all of the necessary information to carry out the task successfully. There are a number of important spatial relations which are missing from h_G (but which are present in h_S). Nevertheless, in combination with the most-specific clause h_S a correct solution to pulling tasks can still be generated.

The final most-specific clause h_S is shown in Figure 5.14. This hypothesis is still quite lengthy because it is a *characteristic* description of the positive examples, rather than a *discriminative* one (as in the case of h_G). It therefore contains some irrelevant terms like `shorter(TubeBack,TubeRight)`, which describes the fact that tubes are longer than they are wide.

```
tool_pose_S( Tool, Box, State):-

% static literals:

attached( Tool, Hook),                                  num_attachments( Hook, 0),
num_attachments( Tool, 1),                              num_attachments( TubeLeft, 1),
num_attachments( Box, 0),                               num_attachments( TubeRight, 1),
longest_component( Tool),                               num_attachments( TubeBack, 0),
shorter( Box, Tool),                                    narrower( Tool, TubeBack),
shape( Tool, sticklike),                                shorter( Hook, Tool),
shape( Box, boxlike),                                   shorter( TubeBack, Tool),
closed_tube( Tube, TubeLeft, TubeRight, TubeBack),      shorter( Box, TubeLeft),
attached_side( Tool, Hook, Side),                       shorter( Hook, TubeLeft),
attached_side( TubeLeft, TubeBack, right),              shorter( Box, TubeRight),
attached_side( TubeRight, TubeBack, left),              shorter( Hook, TubeRight),
attached_end( Tool, Hook, front),                       shorter( TubeBack, TubeRight),
attached_end( TubeLeft, TubeBack, front),               shorter( TubeBack, TubeLeft),
attached_end( TubeRight, TubeBack, front),              shorter( Box, TubeBack),
attached_angle( Tool, Hook, right_angle),               shorter( Hook, TubeBack),
attached_angle( TubeLeft, TubeBack, right_angle),       shape( TubeLeft, sticklike),
attached_angle( TubeRight, TubeBack, right_angle),      shape( TubeRight, sticklike),
attached_type( Tool, Hook, end_to_end),                 shape( TubeBack, sticklike),
attached_type( TubeLeft, TubeBack, end_to_end),
attached_type( TubeRight, TubeBack, end_to_end),

% dynamic literals:

in_gripper( Tool, State),                   in_tube_side( Hook, Tube, Side, State),
touching( Tool, Box, Side, State),          at_right_angles( Hook, TubeLeft, State),
touching( Hook, Box, back, State),          at_right_angles( Hook, TubeRight, State),
in_tube( Box, Tube, State),                 at_right_angles( Tool, TubeBack, State),
in_tube_end( Box, Tube, _, State),          parallel( Tool, TubeLeft, State),
in_tube_side( Box, Tube, Side, State),      parallel( Tool, TubeRight, State),
in_tube( Hook, Tube, State),                parallel( Hook, TubeBack, State).
in_tube_end( Hook, Tube, _, State),
```

Figure 5.14: The final most-specific hypothesis clause h_S, split into static and dynamic components.

The dynamic (spatial) literals in h_S which refer directly to the tool (and are therefore involved in pose selection) are as follows:

```
touching( Tool, Box, Side, State),
touching( Hook, Box, back, State),
in_tube( Hook, Tube, State),
in_tube_end( Hook, Tube, _, State),
in_tube_side( Hook, Tube, Side, State),
```

```
at_right_angles( Hook, TubeRight, State),
at_right_angles( Tool, TubeBack, State),
at_right_angles( Hook, TubeLeft, State),
parallel( Hook, TubeBack, State),
parallel( Tool, TubeRight, State),
parallel( Tool, TubeLeft, State)
```

where `Side` must satisfy `in_tube_side(Box,Tube,Side,State)`. As we have already mentioned, there are a number of redundant literals in this clause. This does not cause any problems because the constraint solver simply ignores constraints which have already been satisfied (recall that the constraint literals are applied one at a time). Typically the subset of (non-redundant) literals actually applied by the constraint solver is much shorter:

```
touching( Hook, Box, back, State),
at_right_angles( Hook, TubeLeft, State),
touching( Tool, Box, Side, State)
```

In fact, following a positive example the learner caches this set of successfully applied spatial constraints and tries to reuse it on subsequent tasks. If a further negative example is encountered then the cached constraints are deleted, and the agent returns to random shuffling of the spatial constraints in h_S. The constraints immediately above were applied successfully for the final four examples in the experiment.

The final most-specific clause h_S also contains the following structural (static) literals which refer directly to the tool:

```
attached( Tool, Hook),
num_attachments( Tool, 1),
longest_component( Tool),
shorter( Box, Tool),
shape( Tool, sticklike),
attached_side( Tool, Hook, Side),
attached_end( Tool, Hook, front),
```

```
attached_angle( Tool, Hook, right_angle),
attached_type( Tool, Hook, end_to_end),
num_attachments( Hook, 0),
narrower( Tool, TubeBack),
shorter( Hook, Tool),
shorter( TubeBack, Tool),
shorter( Hook, TubeLeft),
shorter( Hook, TubeRight),
shorter( Hook, TubeBack)
```

These literals are the ones which are used in the tool selection step. Note that most of the important structural literals have been promoted to the most-general clause h_G, where they are given top priority in the tool selection process.

The agent retains the learnt versions of both h_S and h_G after the completion of the learning episode. Whilst it seems tempting to collapse the 'version space' and select a single learnt hypothesis for the agent to use in future there are significant disadvantages to this approach. If the most general boundary clause h_G alone is chosen to be the final hypothesis the agent may experience a glut of negative examples, since the guidance offered by the most-specific clause has been lost. Conversely, if the most-specific clause h_S is chosen the agent will likely encounter tasks which the planner cannot find a solution to, because the hypothesis is still too specific. Keeping both h_S and h_G intact allows the agent to continue to learn when further examples are encountered in future.

5.3 Push-tool problem

The push-tool problem, as the name suggests, involves using a tool to push an object in order to solve a problem. In our version of the task the robot is given the goal of obtaining a box in an open-ended tube, and must use a stick-tool to extract it. An illustration of our robot performing the task is shown in Figure 5.15.

As in the pull-tool problem of Section 5.2, a teacher demonstrates performing the task with a suitable tool and then the agent is allowed to experiment. The

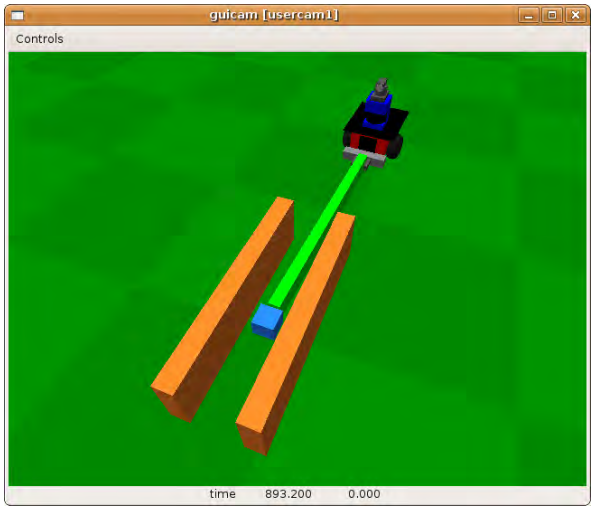


Figure 5.15: The push-tool problem.

types of tool objects available in our pushing experiments are shown in Figure 5.16. Of these tools, the `goodstick` type tool is best for performing the task, and this is the tool chosen for the teacher's demonstration.

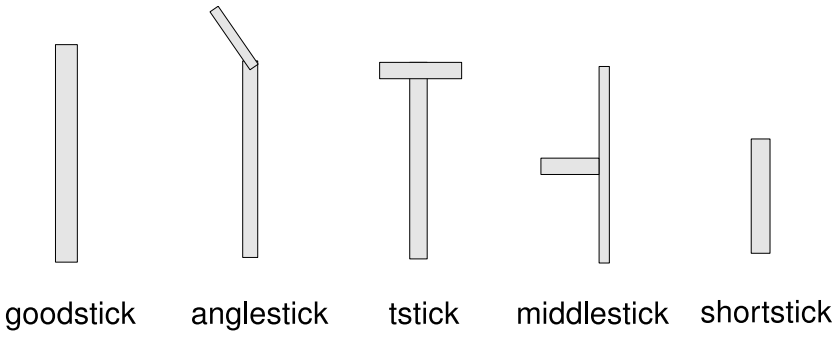


Figure 5.16: Tool types available in the push-tool problem.

The push-tool problem is closely related, at least in an abstract sense, to the pull-tool task. It is nevertheless interesting because it involves learning that a different subset of the state literals are relevant to performing the task correctly. In the pull-tool task the predicate `shorter(A,B)` is usually irrelevant to performing the task (in our experiments the shortest hook sticks were sufficiently long to reach the box). In the push-tool task, however, learning that the tool must satisfy

`shorter(Tubewall,Tool)` (ie. be longer than the tube) is a critical distinction to make.

5.3.1 Background knowledge

The background knowledge predicates provided to the agent for this task are almost identical to those used in the pull-tool task earlier in this Chapter. The one exception is that the `closed_tube(Tube, TubeLeft, TubeRight, TubeBack)` which names the components of the closed tube is replaced with an analogous predicate `open_tube(Tube, TubeLeft, TubeRight)`. The abstract actions defined in the agent's background knowledge are unchanged from the pull-tool task.

5.3.2 Learnt action and tool pose concept

The learnt action model for the push-tool action is:

```
push_from_tube(Tool,Box,Tube)
PRE:   in_tube( Box, Tube),
       in_gripper( Tool),
       gripping,
       tool_pose( Tool, Box)
ADD:   -
DEL:   in_tube(Box, Tube)
```

The learner typically requires 10-15 examples before it is able to consistently solve new versions of this task. Learning is often a bit faster than in the pulling task due to the fact that there are fewer object components involved in the clause (the open tube does not have a `tube_back` part and the tool usually does not have a hook). The learnt most-general `tool_pose` predicate from a typical experiment is:

```
tool_pose_G( Tool, Box, State) :-
        num_attachments( Tool, 0),
        open_tube( Tube, TubeLeft, TubeRight),
        shorter( TubeRight, Tool),
```

```
parallel( Tool, TubeLeft, State),
touching( Tool, Box, front, State).
```

Note that the learner has correctly identified the most important structural properties of the tool — that it should have no attached hooks, and be longer than the tube. It has also learnt to place the tool so that the Box is at the front end of the tool, and the tool is parallel to the tube walls.

In this problem the constraint `parallel(Tool,TubeLeft,State)` is more important than one might initially suppose. Our motion planner does not take account of the fact that if the box collides with the wall it will often start sliding, rather than stop. Thus non-parallel poses for the tool lead the motion planner to predict a collision will occur when it tries to push — as a result, no valid pushing path is returned and the action fails.

The most-specific clause is shown in Figure 5.17. However, most of the interesting predicates have made their way into the most-general clause above.

5.4 Ramp tool problem

The ramp tool problem involves using a ramp to help the agent reach an object on a raised platform. It is analogous to the problem of using a ladder to get up on to a higher surface. Figure 5.18 shows the agent using a ramp in this manner in order get up onto a platform and pick up a box.

This is essentially the same problem that was faced by Shakey the robot (Nilsson, 1984), except that Shakey was supplied with all of the required background knowledge needed in order to complete the task. Our agent is, of course, not given a `climb_ramp` action and must learn the action by watching the teacher and then experimenting in the world.

A selection of the available ramp tools used in our experiments is shown in Figure 5.19. The key properties of a good ramp tool are that it must reach the height of the platform, should not be too steep, and should be wider than the robot. The agent must also, of course, learn to place the ramp in a pose where it is lined up properly with the edge of the platform.

```
tool_pose_S( Tool, Box, State):-

% static literals:

num_attachments( Tool, 0),                          narrower( Tool, TubeRight),
num_attachments( Box, 0),                           narrower( Box, TubeLeft),
longest_component( Tool),                           narrower( Box, TubeRight),
shorter( Box, Tool),                                shorter( TubeLeft, Tool),
shape( Tool, sticklike),                            shorter( TubeRight, Tool),
shape( Box, boxlike),                               shorter( Box, TubeLeft),
open_tube( Tube, TubeLeft, TubeRight),              shorter( Box, TubeRight),
attached_type( TubeLeft, TubeRight, gap),           shape( TubeLeft, sticklike),
num_attachments( TubeLeft, 1),                      shape( TubeRight, sticklike)
num_attachments( TubeRight, 0),
narrower( Tool, TubeLeft),

% dynamic literals:

in_gripper( Tool, State),                   in_tube_side( Tool, Tube, _, State),
touching( Tool, Box, front, State),         parallel( Tool, TubeLeft, State),
in_tube( Box, Tube, State),                 parallel( Tool, TubeRight, State).
in_tube_end( Box, Tube, _, State),
in_tube_side( Box, Tube, _, State),
in_tube( Tool, Tube, State),
```

Figure 5.17: A typical most-specific hypothesis clause h_S learnt for the push-tool problem, split into static and dynamic components.

5.4.1 Background knowledge

For the ramp problem we augment the set of state predicates in the agent's background knowledge with the following additional predicates:

- `shorter_height(A,B)`: True if A is shorter (in height) than B.
- `same_height(A,B)`: True if A is approximately the same height as B.
- `pitch(A,Slope)`: Gives the slope of an inclined object, where Slope can take on values `zero`, `shallow`, `moderate`, `steep`, or `vertical`.
- `on(A,B)`: True if A is on B, where B can be another object or the floor (eg. `on(robot,floor)`).
- `colour(A,Colour)`: Irrelevant to solving the task, but added as an addtional 'distractor' predicate.

The abstract actions for moving objects and recognising `goto` and `carry` actions are the same as for the previous problems in this chapter. However, we

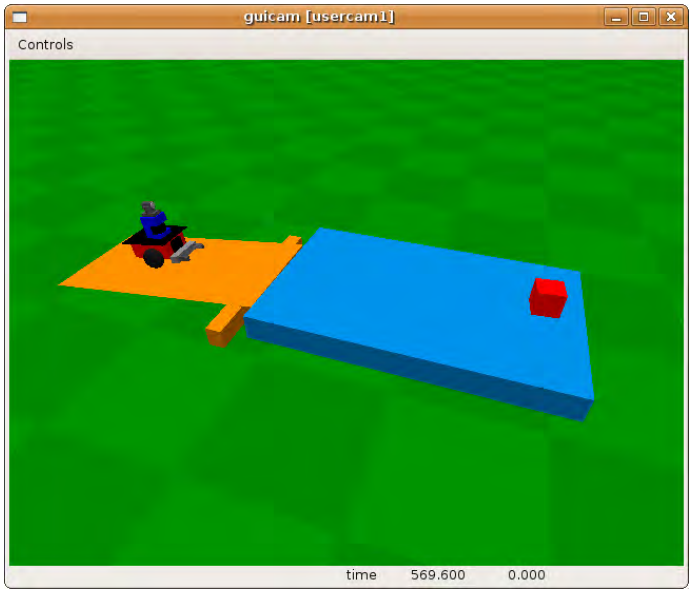


Figure 5.18: The ramp problem. The agent must learn to identify the properties of a useful ramp object, and how it should be correctly positioned.

modify the `put_in_gripper(Obj)` definition to reflect the fact that the robot must be on the same surface (the platform or the floor) as the object in order to pick it up. The action model is therefore:

```
put_in_gripper(Obj,Surface)
PRE     on(Obj,Surface),
        on(robot,Surface),
        empty_gripper,
        ¬gripping,
ADD     in_gripper( Obj)
DEL     empty_gripper
PRIMTV  fwd,back,rotleft,rotright
MOVING  robot
```

5.4.2 Learnt action and tool pose concept

From the explanation of the teacher's demonstration, the agent learns the following simple tool use action model:

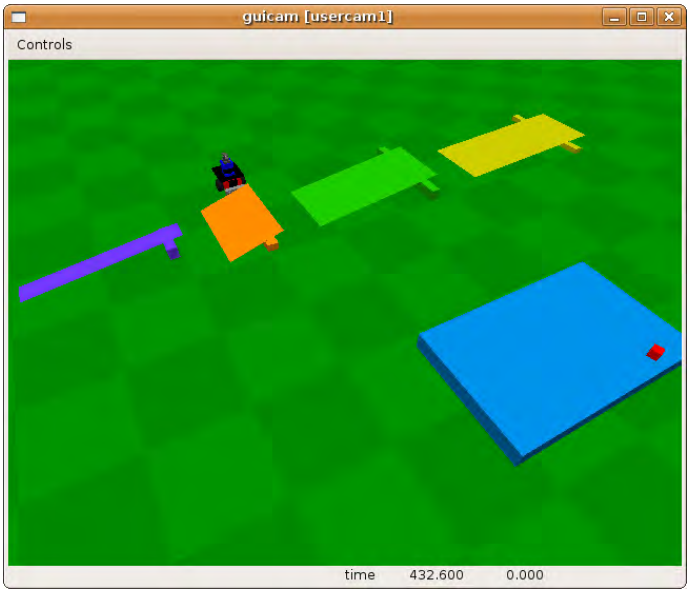


Figure 5.19: Some of the ramp tools available for performing the task. The best ramp tool in this case is the green one — the others shown are too narrow, too steep, or do not reach high enough to access the platform.

```
climb_ramp( Ramp, Platform)
PRE:    ¬on( robot, Platform),
        tool_pose( Ramp, robot)
ADD:    on( robot, Platform)
DEL:    -
```

The sub-goal of this action is to move the robot up on the platform where `on(robot, Platform)` is true. This enables the `put_in_gripper(Obj)` action so that the agent can achieve the goal of picking up the box.

The required properties and spatial pose of the ramp are encapsulated in the `tool_pose` predicate. In our experiments the agent typically learns to solve the task consistently after around 9–13 examples. The most-general tool pose clause learnt during one of these experiments is as follows:

```
tool_pose_G( Ramp, robot, State) :-
        touching( Ramp, Platform, front, State),
        same_height( Ramp, Platform),
```

```
tool_pose_S( Ramp, robot, State):-

% static literals:

attached( Ramp, RampBase),
num_attachments( Ramp, 0),
shorter_length( robot, Ramp),
shape( Ramp, flat),
pitch( Ramp, shallow),
pitch( robot, zero),
num_attachments( Ramp, 1),
num_attachments( RampBase, 0),
narrower( Ramp, Platform),
narrower( Ramp, RampBase),
narrower( robot, Platform),
narrower( robot, Ramp),
shorter_height( Ramp, robot),
shorter_height( RampBase, Ramp),
shorter_height( Platform, robot),
same_height( Ramp, Platform),
shorter_length( robot, Ramp),
shorter_length( robot, RampBase),
shorter_length( RampBase, Ramp),
shorter_length( robot, Platform),
pitch( RampBase, zero),
pitch( Platform, zero),
shape( Ramp, flat),
shape( RampBase, sticklike),
shape( Platform, boxlike)
colour( Ramp, X),
colour( RampBase, X),
colour( Platform, Y),

% dynamic literals:

touching( Ramp, Platform, front,
State),
touching( robot, Ramp, front, State),
onaxis( robot, Platform, State),
onaxis( robot, Ramp, State),
onaxis( RampBase, robot, State),
onaxis( Ramp, Platform, State),
onaxis( RampBase, Platform, State),
at_right_angles( RampBase, robot, State),
at_right_angles( Platform, RampBase, State),
parallel( robot, Platform, State),
parallel( robot, Ramp, State),
parallel( RampBase, Platform, State),
parallel( Ramp, Platform, State).
```

Figure 5.20: The most-specific hypothesis clause h_S learnt on a typical experimental run of the ramp problem, split into static and dynamic components.

```
pitch( Ramp, shallow),
narrower( robot, Ramp).
```

Interestingly, the agent has learnt three correct structural predicates describing the ramp tool pose state, but only one spatial. It has identified that a ramp with a `shallow` slope is necessary, that the ramp should be the same height as the platform, and wider than the robot. However only the `touching` spatial predicate has appeared in the most-general clause. This spatial constraint alone is not sufficient to produce correct placement of the ramp, as there are many primitive states in which the front of the ramp is touching the platform that are not good poses for getting onto the platform.

However, as we saw in the pull-tool problem, the agent is still able to consistently solve the task because it always applies as many spatial constraints from h_S as possible. In this case h_S (shown in Figure 5.20) contains the following non-redundant spatial constraints which are always able to be applied (since any conflicting spatial constraints have already been eliminated):

```
onaxis( Ramp, Platform, State),
parallel( Ramp, Platform, State)
```

Once h_S is reduced to a point where there are no conflicting spatial constraints then all of the constraints will be applied. Learning therefore levels off because our agent never deliberately chooses to apply a smaller subset of constraints than is possible — which would be required in order to obtain additional negative examples to force h_G to be refined.

The justification for our approach is that the agent does not need to produce a compact description of the hypothesis in order to perform the task correctly. However, we note that the agent could choose to engage in deliberate exploration of negative examples once learning has levelled off. This would involve generating experiments which satisfy only a subset of the most-specific constraints. This would be an interesting topic for future work.

5.5 Discussion

We finish our evaluation of our tool-learning agent with a brief discussion of some of the advantages and limitations of our approach.

Importance of an underlying geometric model

One of the reasons our agent is able to learn from relatively few examples in this domain is that it takes advantage of its constraint solver and collision model to rule out potential hypotheses without needing to carry out a physical experiment. The pose selection algorithm effectively tests out a number of candidate hypotheses in the version space as it tries to generate a solution pose. Recall that at each step of the pose selection algorithm the learner adds in a new spatial constraint

literal (creating a new hypothesis to test) and then solves for a solution. If the solution produces a collision pose then that literal is discarded. By allowing the constraint solver to check whether the current working hypothesis is invalid (due to collisions), we are able to search through many more candidate solutions without needing to physically experiment.

Comparison to a top-down approach

In contrast to the top-down approach used by many action model learners (eg. (Benson, 1996; Pasula et al., 2004)), our approach to learning is well suited to the tool use domain. Good performance in the tool domain involves learning lengthy hypotheses which constrain both the spatial arrangement and the structural composition of tools. It is common for a target hypothesis to involve 8 or more literals.

Learning long chains of literals in a top-down manner is difficult, because in order to consider very specific hypotheses one must first eliminate all the more general ones. When only a few examples exist there are many short clauses which may be consistent with the observations. This can lead to a considerable amount of additional experimentation being necessary in order to rule out the shorter hypotheses.

In this work we are focused on having an agent which can learn quickly in a primarily bottom-up manner, based on the lgg. Although the lgg may contain a number of redundant terms, it contains all of the information necessary for the agent to solve the task and the agent can quickly learn lengthy clauses. The problem then becomes that the hypothesis is often *too* specific to be applied to the situation at hand. In this thesis we have presented some useful exploration methods to help overcome this difficulty, by using a version space type representation and using a constraint solver during hypothesis testing.

Noise

It should be noted that our experiments have contained few noisy examples. This is a consequence of the task domain and the perfect state information supplied to the agent. Nevertheless the learning algorithm is indeed capable of dealing with noise in the examples. The main loop which calculates the most-specific clause h_S

operates in an identical manner to GOLEM (at the high level, though not at the level of the lgg). It therefore handles noisy examples in the same way: a "false positive" example will not be included in the most-specific lgg if it causes the clause to lose coverage. False negative examples are accounted for by allowing a clause to cover a certain proportion of negative examples — set by a user-defined parameter. Of course, more noisy examples would increase the number of examples required to learn the hypothesis.

Dependency on user-defined predicates

A limitation of our current approach is that it relies too heavily on user-defined background predicates and constants. For example, we gave the learner five different values for describing the slope of the ramp. In general this approach will not work, because firstly it relies on the user to segment the world well (a very difficult task), and secondly because a single segmentation of slopes, for example, is unlikely to be useful on a wide variety of tasks and objects.

One likely solution to this problem would be to modify the learning algorithm to allow 'lazy' generation of constants during learning. That is, the learner would set the value of the constants in such a way as to maximise coverage over the examples. This could be implemented in the lgg by allowing two numerical constants to be generalised to a range. For example, the lgg of `slope(ramp1,0.1)`, and `slope(ramp2,0.15)` would become `slope(Ramp,0.1-0.15)`. This would allow the agent to generate its own constants which better cover the observed data.

Chapter 6

System architecture and implementation

In this chapter we describe in detail how the various components of our agent are implemented. We begin by illustrating the robot and simulator used in our experiments, and then describe the overall agent architecture and the main control loop. We finish by describing how each system component is implemented, including some low-level algorithms not already described in Chapter 4.

6.1 Robot and simulator

6.1.1 Robot

The agent architecture presented in this thesis is intended to be sufficiently general that it can be applied to any robot. In this research our tool-using robot platform is a Pioneer 2DX with a vertical lift and gripper attachment (see Figure 6.1). Although the Pioneer robot is limited to very simple manipulation, it can perform sufficiently well to demonstrate that the approach we have adopted for tool learning works. Indeed, agents with limited effectors often have the most to gain by employing tools to overcome their shortcomings.

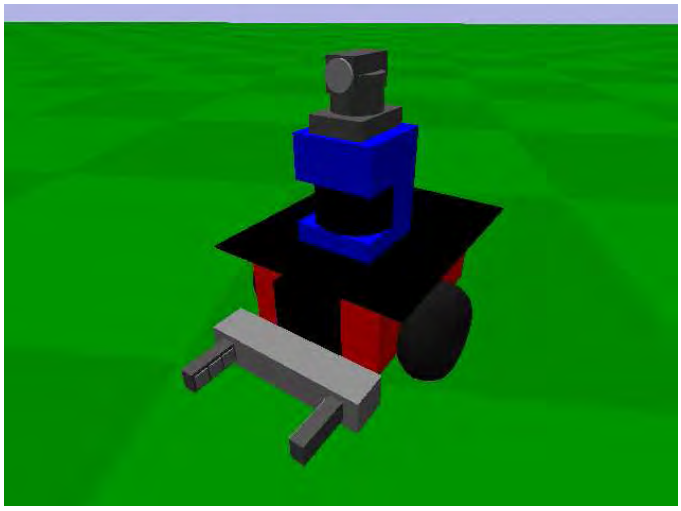


Figure 6.1: Pioneer 2 robot used in our experiments.

6.1.2 Simulator

Our robot tool use experiments are carried out in a three-dimensional rigid-body simulator. This allows us to test ideas and carry out experiments much more rapidly than would be possible in the real world. The simulator used is the Gazebo robot simulator (Koenig and Howard, 2004) which has been developed as part of the well-known Player/Stage (Gerkey et al., 2003) suite of robot control and simulation tools. The experiments described in this thesis were carried out using Gazebo v0.7 and Player v2.0.

The Gazebo simulator is built on top of the Open Dynamics Engine (ODE) (Smith, 2006) physics engine, which simulates rigid-body dynamics in three dimensions and incorporates a collision detection engine. ODE is designed to simulate articulated rigid body structures — that is, structures composed of a collection of rigid bodies held together by a variety of joints. It is unable to simulate non-rigid bodies such as ropes, liquids or pieces of paper.

The robot in our simulator is controlled through the *Player* network interface: a series of clients (one for each sensor/effector) which connect to a central Player server. The Player server reads the current state of the simulator and transmits commands from the robot clients to the simulator. The advantage of the Player

"middle-man" is that client-side robot control code uses the same interface API whether it is communicating with a simulated robot or a real-world robot — this makes it simpler to translate code from the simulation on to a real-world robot.

World specification

The objects, agents, light sources, and observer cameras present in the Gazebo simulator – and their initial configuration – are described in a *world file* written in a simple xml format. An example entry in the world file corresponding to a simple "stick" initially located at position x, y, z with orientation (roll,pitch,yaw) r, p, y is:

```
<model:SimpleSolid>
  <id>stick7</id>
  <xyz>2.400 4.580 0.000</xyz>
  <rpy>0.0 0.0 0.35</rpy>
  <shape>box</shape>
  <size>0.80 0.08 0.12</size>
  <color>1.0 0.0 0.0</color>
  <mass>0.5</mass>
</model:SimpleSolid>
```

Composite objects can be built by adding extra boxes, cylinders, or spheres as sub-models of an existing model. A t-shaped stick tool for example consists of a "parent" box-shaped model (corresponding to the handle), with one "child" box-shaped model (corresponding to the cross-piece). The pose parameters of child models are given relative to the parent body.

All of the objects present in our simulator experiments are built from the `<model:SimpleSolid>` model type described above. More complex types of objects with moveable joints, such as a gripper, must be specified in models written in C++ code. Robots are built from a collection of such models written in C++. In recent release versions (Gazebo-0.9) of the Gazebo simulator, models with joints and forces can be constructed entirely in xml.

6.2 Running times

The experiments described in Chapter 5 were carried out on a Pentium 4 2.5 GHz machine running linux with 1Gb of RAM. As we shall describe in Section 6.3 the learning modules were all implemented in Prolog (SWI-Prolog) for ease of development and the natural symbolic representation. As a consequence our implementation has emphasised ease of prototyping over speed, and a more efficient implementation in a low-level language such as C would likely be an order of magnitude or two faster. Nevertheless, it is worthwhile presenting some approximate running times here.

Table 6.2 shows example run times for each of the experiments presented in Chapter 5. We have broken the run time down to illustrate the contributions from the various components of the algorithm. The columns of the table refer to the time required to segment the teacher's example, construct a STRIPS explanation from these segments (along with novel action models), compute the lgg of a single pair of examples, compute the most-specific hypothesis h_s and obtain the most-general hypothesis h_g.

Problem	**Segment**	**Explain**	**lgg** $(e1, e2)$	$\mathbf{h_s}$	$\mathbf{h_g}$
pull-tool	6.8	0.2	0.03	4.3	0.8
push-tool	5.3	0.2	0.04	2.7	1.2
ramp-tool	15.1	0.2	0.03	3.8	0.9

Table 6.1: Example run times (in seconds) for the pull-tool, push-tool and ramp-tool problems presented in Chapter 5. The times in each column refer to the time required to: segment the teacher's explanation, construct an explanation from the segmented trace, calculate the lgg for a pair of examples, compute the most-specific hypothesis h_s, and reduce the most-specific hypothesis to the most-general hypothesis h_g.

As is apparent from the table, the most time consuming component of the algorithm is the segmentation of the teacher's example. This is because a large number of data frames need to be read from a file, and object velocities calculated in each. Our implementation of this process in Prolog was convenient but inefficient, and clearly a C-based algorithm would perform considerably better. The ramp-tool problem was slowest at segmentation due to the longer length of the

teacher example in this case, producing a larger example file to be processed.

6.3 System architecture

Our agent architecture is illustrated in Figure 6.2 and is comprised of the following eight modules:

- User interface
- Symbolic planning and execution
- Learning by explanation module
- Learning by trial-and-error module
- Spatial constraint solver
- Motion planner
- Collision detector
- Primitive controller

As shown in the Figure, three different programming languages were used in the implementation of these modules. Prolog was used for the high-level control, planning and learning modules. C++ was the language of choice for low-level control, motion planning and collision detection. Finally, ECLiPSe was used only in the constraint solver module, a purpose to which it was ideally suited. In the remainder of this chapter we give further details on how these modules are implemented.

6.3.1 User interface

The learning system is controlled by the user through a simple high-level user interface written in Prolog. After loading the main control program the user is presented with the following command options:

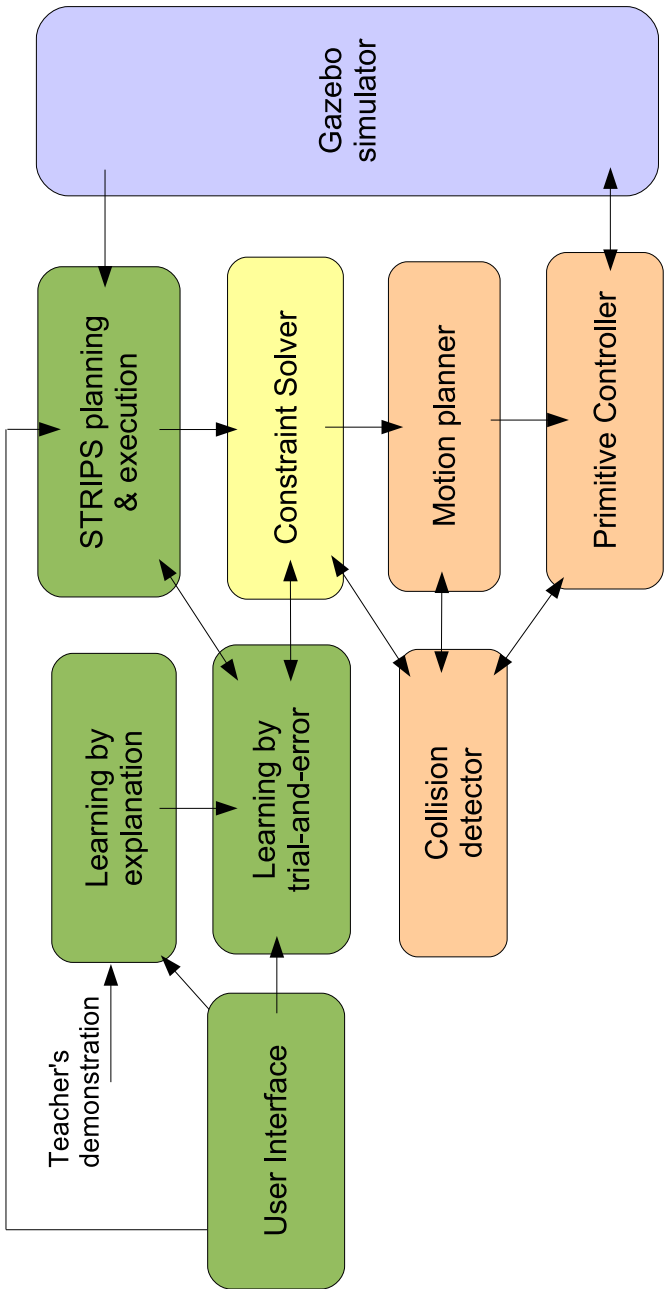


Figure 6.2: System architecture showing component modules and the data flow between them. Green rectangles indicate prolog modules, orange rectangles indicate C++ modules, yellow is an ECLiPSe module, whilst the simulator is shown in purple.

load(Task).	Load a task description and world file.
record_demo.	Allow user to record a demonstration.
learn_demo.	Learn from demonstration.
experiment.	Learn from experimentation.
show.	Show learnt tool action models.

A typical learning session consists of the sequence load(task), record_demo, learn_demo, experiment.

Loading a task definition

A task definition consists of the following, each of which must be specified by the user:

- a task goal
- a tool/object generator file
- an action model definition file
- a learning background knowledge file, containing known state predicate definitions and mode declarations

The goal specifies the desired objective of the agent, which is not possible to achieve without the use of a tool. The aim for all of our tool-use experiments is simply for the agent to obtain a user specified goal object. Thus each of the task descriptions has the same goal: `holding(GoalObject)`, where `GoalObject` is the name of the goal object.

The tool/object generator file defines a set of generalised objects which can appear in a given type of task. For example, in the tube problem the file defines a variety of types of stick objects which can be used by the agent to obtain the reward. The generator program varies the dimensions and composition of each object according to the constraints set out in the file. The tool/object generator file is described in detail in the next section below.

The action model file contains abstract action model definitions of all the actions known to the agent, along with a set of recognition models. Both of these models were described previously in Chapter 3.

Finally, the learning background knowledge file contains all of the information necessary for learning first-order concepts for the given task. First and foremost this consists of a set of predicate definitions which comprise the high-level background knowledge that the agent knows about the world. These predicates are used to interpret the primitive state and provide an abstract description of the current state of the world which can be used by the symbolic planner. In addition to the predicate definitions, the learning file also contains mode declarations used by the relational concept learner in the trial-and-error learning module.

Tool and object generation

Each time the agent succeeds at achieving the goal we present it with a new version of the task. For the tube problem, this might involve a tube of different length and/or width and the goal object located at a different position within the tube. In addition, the set of objects which are available to be used as tools are varied in their length, width, and composition.

The kinds of objects and tools which may appear in a task are specified in a tool/object generator file. This file consists of a set of object type definitions which describe the way in which kinds of objects are constructed, and the ways in which they are allowed to vary. Each type definition specifies:

- A list of component parts
- Allowed range of dimensions for the component parts
- An allowed range of angles and locations for the (fixed) joints which connect these parts
- The allowed initial locations for the object (or alternatively the allowed initial region)

A generate_world program written in Prolog is used to translate a set of type definitions into a world file which can be input into the Gazebo simulator. The generator randomly creates a set of objects which fit the parameters specified in the object type definitions.

One example of a type of object in the tube problem is a `hookstick`, which consists of a single "handle" with a hook at the end of the handle. The definition

of a `hookstick` type object in the tube problem task file is as follows:

```
object_type( tstick).
has_part( tstick, handle).
has_part( tstick, hook).
length( handle, 0.5:1.5).
length( hook, 0.02:0.40).
width( handle, 0.03:0.20).
width( hook, 0.03:0.15).
height( handle, 0.05:0.05).
height( hook, 0.05:0.05).
attach( handle, hook, 0.9:1.0, 0.0:0.0, 90:90).
```

This object type definition says that the length of the handle must be between 0.5m and 1.5m long, with a width between 3cm and 20cm. The `attach( handle, hook, 0.9:1.0, 0.0:0.0, 90:90)` literal says that the hook should be attached to the handle at a point which is between 90-100% along the length of the handle, and 0% along the length of the hook, at an angle of 90 degrees.

The tool/object generator file also defines the number of objects of each type which should appear, and their location. In our experiments we supply the agent with a choice of six different objects each time a new task is generated. The objects are arranged near the starting location of the robot, spaced evenly to ensure that they can be accessed easily and do not get in the way of each other.

Recording a demonstration

A demonstration of tool-use consists of the user driving the robot around to show how a task can be accomplished using a suitable object as a tool. Before the demonstration begins the user loads a task file and generates a random version of the task.

The robot is controlled by the user via the user interface, using the Gazebo GUI wxgazebo to display an image of the scene. The user is able to execute the same abstract actions which are posessed by the learner agent, in addition to the new tool action which is the target of the learning process. Alternatively the user

may choose to use the Gazebo GUI interface to control the robot directly with primitive commands.

As the user drives the robot around the agent records primitive world state snapshots at fixed intervals of time (we use 0.1 seconds in our experiments). For each snapshot the pose of every object in the world is recorded. The agent does not have access to view the commands executed by the user, and so must infer its behaviour by watching changes in the primitive state. The user does not give any high-level information to the agent about the demonstration — either its goal or how it was achieved.

6.3.2 STRIPS Planner

For the planner module we use an off-the-shelf implementation of the Fast Forward (FF) planning system (Hoffmann and Nebel, 2001). Fast Forward, as the name suggests, is a forward search planner which searches through the state space using a goal-distance heuristic which ignores delete lists. It was the most successful automatic planner at the AIPS-2000 planning competition (*ibid.*).

The FF planner takes as input a set of action models and a problem definition written in standard Planning Domain Definition Language (PDDL) syntax (McDermott, 2000). We have written a parser which translates our abstract action models from Prolog into a corresponding PDDL file. A second parser captures the output from the planner and translates it back into Prolog so that it can be understood by our agent.

6.3.3 Motion planner

For the motion planner module we have implemented a version of the *RRT-Connect* (Kuffner and LaValle, 2000) algorithm, a Rapidly-exploring Random Tree (RRT) motion planner. The RRT algorithm is well-suited to robotics applications where the dimensionality of the search space is often high, and the algorithm has been successfully used to plan complex motions of humanoid robots in cluttered environments (Kuffner et al., 2003; Hirano et al., 2005). RRT works by sampling poses in real space (as opposed to building a configuration space) and employs a heuris-

tic which causes it to aggressively and efficiently explore unvisited regions of the state space. The *RRT-Connect* version of the algorithm used in this thesis involves two search trees which fan out from the goal and the initial state respectively, and attempt to connect somewhere in the middle.

Our implementation of *RRT-Connect* is written in C++ and takes three inputs:

- A set of fixed "background" objects in specific poses.
- A set of manipulated objects in a fixed relative pose (eg. a pulling tool attached to the robot).
- A set of permissible displacements in real space, representing the allowed motion primitives.

As explained in Chapter 3 the motion planner is used to generate trajectories to primitive goal states which satisfy the goals of abstract actions. Below we discuss how the constraint solver is able to convert from abstract into primitive goal states.

6.3.4 Constraint solver

Each time the agent wishes to execute an abstract action it must be able to identify a primitive goal state which satisfies the abstract goals of the action — in other words, the abstract goals must be made operational. Consider for example executing the action `put_on(book,table)` which has primary goal `on(book,table)`. The agent must first find a primitive world state in which `on(book,table)` holds. It can then use its motion planner to find a path which leads to this state.

One way of finding a primitive state which satisfies an action sub-goal is to generate a random selection of primitive states and check if any of them satisfy the goal property. This is, however, a terribly inefficient way of finding a suitable goal state. In the case of the sub-goal `on(book,table)` there are an infinite number of states which do *not* satisfy the goal condition, and randomly sampling the space of relative poses between the `book` and `table` objects looking for a goal state would be very slow indeed.

The problem of finding a primitive state which satisfies an abstract goal is a constraint satisfaction problem. Since the constraints (goals) we wish to satisfy are expressed in first-order logic we use the constraint logic programming system

ECLiPSe (Apt and Wallace, 2007) to find primitive solution states. ECLiPSe is able to use the abstract state concept definitions written in Prolog to search efficiently for satisfactory primitive solution states.

The ECLiPSe constraint solver in our system is run as an independent module which is accessed through a socket interface. It receives a set of abstract spatial goals to solve, and sends back a primitive state which satisfies these goal constraints. Within ECLiPSe we are using the interval constraint library ic, which provides constraint propagation and search mechanisms on real valued quantities (in this case object pose coordinates).

Preprocessing state predicates

In order to apply ECLiPSe to our state predicates written in Prolog we must first do some preprocessing to convert Prolog goals into ECLiPSe constraints. The preprocessing step is performed just once, after the state predicates have been defined in Prolog. The file containing Prolog state predicate definitions worldstate.pl is converted to an equivalent ECLiPSe file worldstate.ecl via a small program written in Prolog.

Converting Prolog goals to ECLiPSe constraints involves replacing any arithmetic Prolog *comparison* operators (less than, greater than, equals etc.) which occur in the predicate definitions with corresponding ECLiPSe arithmetic *constraint* operators.

As an example of this process, consider the state predicate `on_axis(ObjA, ObjB, State)` which is true when `ObjA` lies on the axis of `ObjB` in a given primitive state. This abstract state predicate is defined in Prolog as:

```
on_axis( ObjA, ObjB, State) :-
    pose( ObjA, PoseA, State),
    pose( ObjB, PoseB, State),
    relpose( PoseB, PoseA, [X,Y,Theta]),
    close_const(C),
    Y < C,
    Y > -C.
```

where `relpose` is a geometric predicate which calculates the relative pose `[X,Y,Theta]` of `PoseA` relative to `PoseB`. This Prolog predicate is replaced in ECLiPSe with an equivalent predicate containing constraints rather than comparison operators:

```
on_axis( ObjA, ObjB, State) :-
    pose( ObjA, PoseA, State),
    pose( ObjB, PoseB, State),
    relpose( PoseB, PoseA, [X,Y,Theta]),
    close_const(C),
    Y $< C,
    Y $> -C.
```

Here we have replaced the greater-than and less-than comparison operators (> and <) with the corresponding arithmetic constraint operators `$>` and `$<`. In the general case the arithmetic comparisons $<, =<, ==, \backslash=, >=, >$ are replaced by the corresponding arithmetic constraints $\$ <, \$ =<, \$ =, \$\backslash=, \$ >=, \$ >$. In addition the numeric evaluation predicate '**is**' is replaced by $\$ =$. When ECLiPSe encounters these primitive constraints as it executes a predicate, it applies the relevant constraint to numeric variables involved, rather than evaluating for true or false (as Prolog would do). In our predicates the variables represent, or are directly linked to, the poses of the objects we are trying to constrain. ECLiPSe is able to propagate these constraints in an efficient way so that constraints applied to one object are also applied to connected objects.

Finding a solution: An example

The method used to find a ground state which satisfies one or more goals is best illustrated with an example. Suppose we wish to find a primitive state `State` which satisfies `on_axis(stick,robot,State)` so that the stick is lined up on the axis of the robot. To represent the primitive state we set the `State` parameter equal to a list of objects and their corresponding poses. Using a robot-centred coordinate system, with the robot pose fixed at the origin, `State` can be written as:

```
State = [robot:[0,0,0], stick:[X,Y,Theta]]
```

where `X, Y, Theta` are variables representing the `stick` pose coordinates which we wish to constrain.

We begin by limiting the domain of these variables to -5.0 to 5.0 (for X and Y) and -pi to pi (for Theta), where the X and Y values are expressed in metres. The `on_axis( stick,robot,State)` goal can then be solved by simply calling it from within ECLiPSe as follows:

```
[eclipse]:    State = [ robot:[0,0,0], stick:[X,Y,Theta] ],
              [X,Y]::[-5.0..5.0], Theta::[-3.14..3.14],
              on_axis(stick,robot,State).
```

The `::` operator constrains the domain of a variable to the range on the right hand side. ECLiPSe immediately gives the constrained solution (in robot-centred coordinates):

```
X = X{-5.0..5.0}
Y = Y{-0.05..0.05}
Theta = Theta{-3.14..3.14}
```

which indicates that the stick must be positioned along the robot's main axis where the `Y` coordinate is limited to the values $-0.05 < Y < 0.05$ as shown in Figure 6.3. Meanwhile the `X` and `Theta` coordinates are not constrained any further within their defined domains.

This method gives us a range of solutions, expressed as a set of simple constraints on the `X,Y,Theta` coordinates. In this simple example, every point within the range of coordinate values corresponds to a valid solution. In general, however, ECLiPSe does not guarantee that every point will give a solution — only that at least one solution lies somewhere within the range indicated. To find these solutions requires committing to particular choices for one variable or another, and then backtracking where necessary — that is, search. The ECLiPSe built-in predicate `locate` does the heavy-lifting, and simply adding `locate([X,Y,Theta],0.01,lin)` to the end of the above ECLiPSe program will cause ECLiPSe to search for specific solutions within an accuracy of 0.01 (the parameter `lin` tells it to split the domain linearly during search, rather than logarithmically).

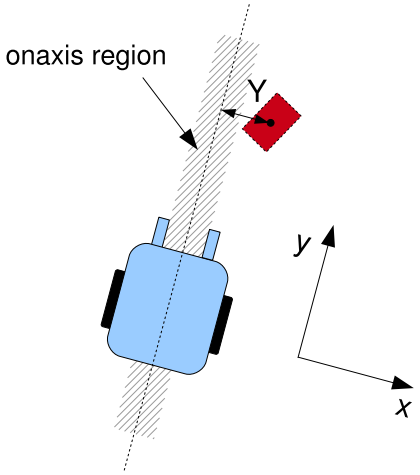


Figure 6.3: Illustration of the `onaxis` predicate.

If we wish to find a solution to `on_axis(stick,robot,State)` in world coordinates – rather than robot-centred coordinates – we can do so by applying a further constraint: that all objects satisfy the same coordinate transformation between world coordinates and robot coordinates. If we let the robot pose (in world coordinates) of the solution be `RobotPose`, and the object pose be `RelPose` (in robot coordinates) or `WorldPose` (in world coordinates) then these quantities must all satisfy the coordinate transformation constraint `relpose( RobotPose, WorldPose, RelPose)`. If the `stick` object is located at, say, `WorldPose=[0.3,1.2,-0.8]` (in world coordinates) then the complete ECLiPSe call to find a solution to `on_axis(stick,robot,State)` is:

```
[eclipse]:    State = [ robot:[0,0,0], stick:[X,Y,Theta] ],
              [X,Y] ::  [-5.0..5.0], Theta ::  [-3.14..3.14],
              WorldPose = [0.3,1.2,-0.8],
              on_axis( robot, stick, State),
              relpose( [Rx,Ry,RTh], WorldPose, [X,Y,Theta]),
              locate( [Rx,Ry,RTh], 0.1, lin).
```

This returns a solution `[Rx,Ry,RTh]` which corresponds to the robot pose in world coordinates.

Algorithm 3 Solve a set of goals using ECLiPSe.

```
solve( +Goals, +RelToolPoses, +BackgrState, -Soln)
```

```
 1: solve( Goals, RelToolPoses, BackgrState, Soln) :-
 2:     create_var_state( BackgrState, VarState),
 3:     restrict_domains( VarState),
 4:     restrict_domains( [robot:[Rx,Ry,RTh]]),
 5:     append( robot:[Rx,Ry,RTh]), RelToolPoses, VarState, State),
 6:     call( Goals),
 7:     get_backgr_constraints( BackgrState, State, [Rx,Ry,RTh], BackgrConstraints),
 8:     call( BackgrConstraints),
 9:     locate( [Rx,Ry,RTh], 0.1, lin),
10:     collision_check( [Rx,Ry,RTh]),
11:     Soln = [Rx,Ry,RTh].
```

Non-physical states: Collision checking

Finding a primitive state solution to a set of goals does not guarantee that the primitive state is a valid state in the real world — in particular, it may imply a collision between two rigid-body objects occupying the same region of space. To avoid producing non-physical states we add a collision check to any solutions produced using the above method.

It should be noted that whilst it is possible to introduce a "no collisions" predicate as an explicit constraint, we do not do so in this work. Rather, we apply the check to any solutions produced after `solve` is called. We decided against using an explicit constraint simply because it would introduce a lot of additional non-linear constraints, require re-writing the code in an ECLiPSe-friendly form, and would add a lot of choice points to the search. In practice, imposing the check after the `locate` predicate is called still allows for backtracking over `locate` when collisions are encountered and was sufficient for our purposes in this work.

6.3.5 Finding a solution: General case

The general form of our `solve` predicate is given in Algorithm 3. The steps

involved in finding a solution using `solve` are as follows:

1. **Construct the inputs:**
   There are three inputs to the `solve` predicate:
   (a) `Goals`: A list of goals to be solved (eg. `on_axis(stick,robot,State)`)
   (b) `RelToolPoses`: A list of object poses describing the (fixed) orientation of the tool(s) relative to the robot at the origin. For example, in the case of a `robot` gripping a `stick` so that it sits at a position 0.5m directly in front of the robot the `RelToolPoses` state would be:
   `RelToolPoses = [robot:[0.0,0.0,0.0], stick:[0.5, 0.0, 0.0]]`
   Note that this state is written in robot-centred coordinates.
   (c) `BackgrState`: A list of all other object poses (ie. other than those in `RelToolPoses`). These poses should correspond to the fixed position of the objects in world coordinates. eg.
   `BackgrState = [tube:[3.4,1.2,0.0], reward:[3.4, 0.6, 0.5]]`
   (d) `SolnState`: A variable for returning the solution state.
2. **Create a variable state:**
   Create a state containing the same objects as those in `BackgrState`, except replace the fixed poses with variables. eg.
   `VarState = [tube:[X1,Y1,Th1], reward:[X2,Y2,Th2]]`
3. **Construct the search state:**
   The search state `State` is defined by appending the `RelToolPoses` to the `VarState`. This corresponds to the state of the world in robot-centred coordinates, where the objects other than the robot and tool are given variable poses which we wish to constrain by the list of `Goals`.
4. **Set search state equal to state parameter in goals:**
   The last parameter of each dynamic goal is an empty state variable. This variable in each goal, must be set equal to the search state defined in the previous step. This allows the goals to impose their constraints on the solution.
5. **Call the goals:**

The goals are executed, which applies the constraints implied in the definitions of the goal predicates.

6. **Constrain to fit background state:**
   At this point, the `State` variable contains goal constraints applied in a robot-centred coordinate system. We now apply a further set of constraints to ensure that when the robot-centred solution is transformed into world coordinates it is equal to the poses defined in `BackgrState`. This is done by creating a set of goals of the form `relpose( [Rx,Ry,RTh], WorldPose, RelPose)` for each object in the background state, and executing the goals to apply the constraints.

7. **Locate solutions:**
   The penultimate step is to search for solutions which satisfy all of the constraints. This is done using the built-in search predicate `locate`. We use a `linear` division of the search space when calling this predicate, and search for solutions to within an accuracy of 0.01 (the second and third parameters of `locate`).

8. **Check for collisions:**
   The final step is to check that the solution given by `[Rx,Ry,RTh]` is a valid state by calculating the complete state and calling the collision checker. If collisions occur then we backtrack to other solutions produced by `locate`.

6.3.6 Collision detector

The agent is provided with a simple geometric model of each object in the world as part of its background knowledge. In the case of a stick object, for example, this would simply consist of its length, width and height. The collision detection module uses these geometric models to determine whether any given pair of objects is colliding.

The collision detector is employed by a number of higher-level modules, as shown in Figure 6.2. The *constraint solver* module uses it to check whether its solutions to logical constraints are valid poses. The *motion planner* module uses it to find collision-free paths to a given sub-goal. Finally, the *primitive controller*

uses to detect when a collision has occurred during execution.

The collision detection engine used in our work is the C++ library SWIFT (Ehmann, 2001; Ehmann and Lin, 2000). SWIFT is a simple library for collision detection and distance computation for rigid three-dimensional polygonal objects. Any alternative collision detection library would work just as well, as we have made no alterations to the code.

6.3.7 Primitive controller

The primitive controller module is a straightforward pd-controller used to drive the robot along a path supplied by the motion planner. This module is implemented entirely in C++. The desired robot path is input to the controller as a list of intermediate poses which must be passed through on the way to the final goal pose. After each intermediate pose has been attained (approximately) the controller sets the subsequent intermediate pose as the new target.

In cases where an unexpected collision occurs, causing the robot to become stuck, the controller returns a "fail" signal to the abstract action (eg. `put_in_gripper`) which called it. The STRIPS planner is then able to replan, the motion planner is called once again, and a new path is generated. In rare cases in our experiments where the agent was unable to recover from getting stuck the simulator was simply reset. However, if the agent was in the process of using the tool when it became stuck, a negative example of the tool pose state was first recorded.

6.3.8 Learning by explanation module

The learning by explanation module takes as input the teacher's demonstration trace and outputs an explanation, along with any novel action models which were needed to complete the explanation. The new action models are then refined through additional trial-and-error based learning.

The learning by explanation module is written entirely in Prolog and consists of three primary components:

- segmentation of the teacher's trace
- segment labelling

- construction of action models for novel action segments

The approach to each of the above components of the explanation-based algorithm has been given in Chapter 4, and illustrated in Chapter 5. Below we give some more detailed pseudo-code and some additional comments on its implementation.

Segmentation

High-level pseudo-code for the segmentation process is given in Algorithm 4. The first step in the segmentation algorithm is the thresholding of object velocities observed in the teacher's example trace. As described in Chapter 4, a hysteresis-style threshold is used to avoid flickering between stationary and moving states. Detailed code is not given here as the process is straightforward.

The thresholding process is followed by creating a list of segment boundaries occurring during the trace. A segment boundary is placed at each point in the trace at which any object stops or starts moving. A segment boundary is represented in the list by a triple of the form (*frame*, *object*, 'start'/'stop'), where *frame* is the frame number at which the transition occurs, *object* is the name of the object, and 'start' or 'stop' notes whether the object started or stopped moving at that point.

The initial list of segment boundaries is then sorted (into chronological order) and used to build a sequence of segments. A segment is represented by a pair (*frame*, *objects*) where *frame* is the frame at which the segment begins, and *objects* is a list of all the objects which are moving within the segment. The end point of the segment is defined implicitly by the start point of the next segment in the sequence. The objects which are moving in the segment are determined by keeping track of a list of moving objects (*movingobjs* in Algorithm 4) and updating it each time a segment boundary is traversed — meaning an object has started or stopped moving.

An important part of the process of creating the segments list is to merge segment boundaries which lie very close to one another. As discussed in Chapter 4, it is common for objects to start or stop moving at almost the same time. This produces a series of closely spaced boundaries, which our code treats as a single 'combined' boundary for the purposes of building segments. It does this by

Algorithm 4 Segmentation of the teacher's example.

```
SEGMENT_TRACE( trace)
 1: segBoundaries ← {}
 2: for each object obj do
 3:    for each frame in trace do
 4:       if obj started moving in frame then
 5:          segBoundaries ← segBoundaries + (frame, obj, 'start')
 6:       else if obj stopped moving in frame then
 7:          segBoundaries ← segBoundaries + (frame, obj, 'stop')
 8:       end if
 9:    end for
10: end for
11: Sort segBoundaries by frame
12: last ← 0
13: movingobjs ← {}
14: segments ← {}
15: for each (frame, obj, type) in segBoundaries do
16:    if type='start' then
17:       movingobjs ← movingobjs + obj
18:    else
19:       movingobjs ← movingobjs − obj
20:    end if
21:    f ← frame
22:    if frame − last < mergethreshold then
23:       loop
24:    else
25:       segments ← segments + (f, movingobjs)
26:       last ← f
27:    end if
28: end for
29: segments ← remove 'paused' segments in segments
30: return segments
```

ignoring segment boundaries which lie within *mergethreshold* of the most recently started segment.

The final step in the segmentation process is to remove any segments in which no objects are moving (which we refer to as "paused" segments). This occurs when the robot stops to, for example, turn and move in a new direction. Paused segments contain no interesting information and can be safely removed without affecting the explanation of the teacher's example.

The end result of the segmentation process is a list of segments, represented by a series of (*frame*, *objects*) pairs, where *objects* is a list of objects moving in the segment which began at frame *frame*. It should be noted that the above algorithm assumes that the velocities of objects determined by the observing agent are not excessively noisy (to avoid spurious segments being created). To deal with the case of a noisy state signal we have conducted preliminary experiments with anisotropic smoothing (Rao et al., 2002) and it appears to be a promising approach. Smoothing was ultimately not necessary in our simulations due to the lack of noise in the simulator data. However, when moving on to a real-world robot (as described in Chapter 8) some form of velocity smoothing would need to be added to the implementation.

Labelling segments

The next step in the explanation-based learning module is to label the segments produced by the segmentation algorithm. Pseudo-code for this process is given in Algortihm 5, and a detailed explanation and illustration of this algorithm were presented in Chapter 4 and Chapter 5 respectively.

The algorithm works in a straightforward manner, by looping through each segment identified in the teacher's example trace and attempting to recognise the action(s) being performed within the segment. Each segment can be labelled by:

- a single known action
- a sequence of known actions
- a recognition model (representing a general positioning-type action)
- a novel action

Algorithm 5 Label the segments in the teacher's example.

```
LABEL_SEGMENT( segments)
 1: labelled ← {}
 2: for each seg in segments do
 3:    action ← EXPLAIN_SEGMENT( seg)
 4:    labelled ← labelled + {action}
 5: end for
 6: return labelled

EXPLAIN_SEGMENT( segment)
 1: action ← EXPLAIN_SEGMENT_ACTION( segment)
 2: if action ≠ {} then
 3:    return action
 4: else
 5:    model ← EXPLAIN_SEGMENT_RECOGMODEL( segment)
 6:    if model ≠ none then
 7:       return model
 8:    else
 9:       return 'novelAction'
10:    end if
11: end if
```

The EXPLAIN_SEGMENT(*segment*) procedure in Algorithm 5 carries out this process of labelling a given segment. In order to do so, it calls the sub-procedures EXPLAIN_SEGMENT_ACTION(*segment*) and EXPLAIN_SEGMENT_RECOGMODEL(*segment*). The first of these, EXPLAIN_SEGMENT_ACTION attempts to match the segment with a sequence of one or more actions in the agent's background knowledge (ie. known actions). If this fails, EXPLAIN_SEGMENT_RECOGMODEL is used to try and match the segment to a recognition model. If this, in turn, fails the agent labels the segment as a novel action.

The EXPLAIN_SEGMENT_ACTION(*segment*) procedure (Algorithm 6) works by seeking to find a series of known actions $A_1, A_2, \ldots A_n$ which can be used to "cover"

Algorithm 6 Explain a segment as a sequence of one or more known actions.

EXPLAIN_SEGMENT_ACTION(*segment*)

1: *moving* ← objects moving in *segment*
2: *enabled* ← list of 'enabled' actions in segment, each represented as a tuple (*action, P1, P2, E*), where *P1, P2* and *E* are the frames at which:
   - *P1*: the action's precons become true
   - *P2*: the action's precons cease to be true
   - *E*: the action's effects become true
3: Remove from *enabled* any actions whose action model's object list does not match with *moving*
4: *begin* ← beginning frame of *segment*
5: *end* ← last frame of *segment*
6: *explanation* ← ordered sublist $A_1, A_2, \ldots A_n$ of actions in *enabled* which cover the segment from *begin* to *end* such that:
   - the precondition of A_1 is true at *begin*
   - the effects of A_n are true at *end*
   - for any successive actions A_i and A_{i+1} with tuples $(A_i, P1_i, P2_i, E_i)$ and $(A_{i+1}, P1_{i+1}, P2_{i+1}, E_{i+1})$, the condition $P1_{i+1} < E_i < P2_{i+1}$ is satisfied
7: **return** *explanation*

or "traverse" the segment from beginning to end. The algorithm imposes the constraint that the preconditions of the first action A_1 must be true at the beginning of the segment, and the effects of the final action A_n must be true at the end of the segment. In addition, at the point at which one action A_i ends and the next action A_{i+1} begins, the effects of A_i must be true and the preconditions of A_{i+1} must be true. In many cases the sequence may consist of a single action, whose preconditions are true at the beginning of the segment and whose effects are true at the end.

In order to facilitate the process of matching the sequence of preconditions and effects, the algorithm first produces a list of actions whose preconditions and effects are satisfied at some point during the segment — these actions are said to be *enabled*. For each enabled action three key frames numbers are recorded:

Algorithm 7 Explain a segment with a recognition model.

```
EXPLAIN_SEGMENT_RECOGMODEL( SEGMENT)
 1: moving ← objects moving in segment
 2: for each model in agent's recognition models do
 3:    if moving is not a subset of model.MOVING then
 4:       loop
 5:    end if
 6:    preState ← world state at start of segment
 7:    if model.PRE not satisfied in preState then
 8:       loop
 9:    end if
10:    explanations ← explanations + {model}
11: end for
12: if explanations = {} then
13:    return none
14: else
15:    model' ← model in explanations with the most specific precons
16:    return model'
17: end if
```

- P1: the frame at which the precondition is first satisfied
- P2: the frame at which the precondition ceases to be satisfied
- E: the frame at which the effects are first satisfied

These frames are illustrated in Figure 6.4. By recording a list of enabled actions in the form of a list of tuples (Action, P1, P2, E), the algorithm can find a valid sequence which traverses the segment from beginning to end.

If the agent is unable to match the segment to a sequence of one or more known actions, the algorithm instead tries to match the segment to one of its recognition models. The pseudo code for this process is given in Algorithm 7. In order to be matched to a segment a recognition model must share the same list of moving objects as observed in the segment, and the preconditions must be true at the start of the segment. If more than one recognition model is satisfied, the one with

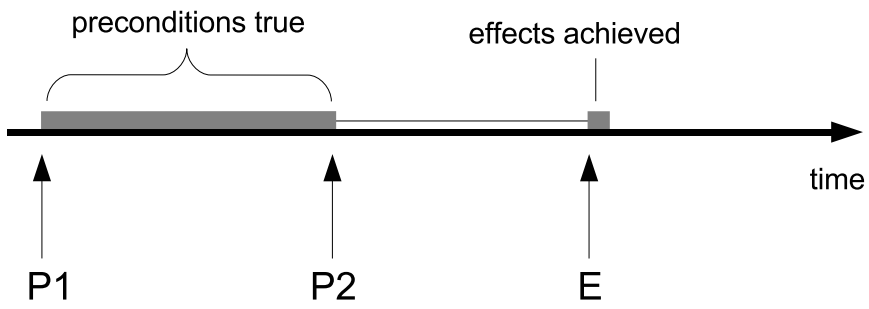


Figure 6.4: Illustration of the quantities P1, P2 and E referred to in Algorithm 6

the most specific preconditions is selected.

Finally, if the agent is unable to match the segment to known action(s) or a recognition model, the EXPLAIN_SEGMENT algorithm labels it as a novel action. A new action model to represent this novel action is constructed in the last step of the explanation-based learning module.

Building novel action models

The final step in the explanation-based learning algorithm involves building new action models to represent the observed novel action. Recall from Chapter 4 that novel tool actions usually consist of a "positioning" action, where the tool is put into place, followed by an "effects" action in which the desired effect of the tool is achieved. Although other types of tool action structures are possible (see Chapter 4) this is the common form of tool action which is addressed in this thesis.

Pseudo-code for building action models for the two components (positioning-step and effects-step) of the novel tool action is given in Algorithm 8. The algorithm takes as input the explanation generated by the segment labelling algorithm described in the previous section, and outputs new action models which are added to the agent's action model database.

The first step of the algorithm is to split the explanation of the teacher's actions into four pieces:

- *actionsBefore*: actions occurring before any recognition model or novel action

Algorithm 8 Building new action models to explain novel segments.

BUILD_NEW_ACTION_MODELS(*labelledSegments*)

1: Split *labelledSegments* into four pieces: *actionsBefore, recognModel, novelAction, actionsAfter*
2: *unsupportedPre* ← unsupported precons in *actionsAfter*
3: *unexplained1* ← unexplained effects in *recognModel* segment
4: *unexplained2* ← unexplained effects in *novelAction* segment
5: Define new positioning-step action model *placeTool* with:
   placeTool.PRE ← *recognModel.PRE*
   placeTool.EFFECTS ← *tool_pose(Tool)*, *unexplained1*
   placeTool.MOVING ← *recognModel.MOVING*
6: Define new effect-step action model *useTool* with:
   useTool.PRE ← net effects of *actionsBefore* and *placeTool*
   useTool.EFFECTS ← *unexplained2*
   useTool.MOVING ← objects moving in *novelAction* segment
7: Replace ground objects by variables in *placeTool, useTool*
8: **return** *placeTool, useTool*

- *recognModel*: the recognition model (the positioning-step)
- *novelAction*: the novel action (the effects-step)
- *actionsAfter*: actions occurring after the novel action

The algorithm identifies any actions in *actionsAfter* which have unsupported preconditions — that is, any actions with preconditions which are not enabled by an earlier action. Likewise, any unexplained effects in the segments corresponding to *recognModel* and *novelAction* are recorded. An unexplained effect is defined as any effect which might support a later action in the explanation, but is not yet accounted for by the existing action models represented in the explanation.

The algorithm uses the lists of unsupported preconditions and unexplained effects to construct new action models for the positioning and effects step of the novel action according to the principles defined in Chapter 4. The primary effect of the positioning action is defined to be the tool pose state literal `tool_pose(Tool)`.

This literal, which is also a precondition of the effects-step, encapsulates all of the additional positional and structural conditions which must be true in order for the tool-use action to succeed. These additional requirements might include restrictions on the dimensions or shape of the tool, along with a detailed description of the relative pose in which it must be placed. A definition of the `tool_pose` literal is learnt through inductive logic programming in the agent's trial-and-error learning module.

6.3.9 Learning by trial-and-error module

The agent's learning by trial-and-error module aims to learn a definition of the structural and positional conditions under which the tool can be used to achieve the desired goals. Whilst some of the novel action preconditions were inferred by the explanation module, the remainder were encapsulated by an undefined `tool_pose(Tool)` predicate. The trial-and-error module learns a definition of this predicate through repeated experimentation and the use of an inductive logic programming learning algorithm.

Pseduo-code for the main loop of the trial-and-error learner is shown in Algorithm 9. As with the learning by explanation module, all of the code in this module is implemented entirely in SWI-Prolog.

The algorithm repeatedly selects a new tool and tries to use it to solve the task. For ease of experimentation the same generic plan is used each time the agent performs the task. Each time a new tool is selected, the agent updates the plan with the newly selected tool. During execution of the plan the agent records the state *learnState* — defined as the world state which exists at the point the novel action effects-step commences. If the novel action succeeds, the agent adds *learnState* as a positive example of the `tool_pose(Tool,State)` predicate. Conversely, if the action failed *learnState* becomes a negative example of `tool_pose(Tool,State)`.

To record an example the agent creates a new label for the state, such as `s7`, and records the new example in the examples file. For instance, if `s7` was a positive example involving `stick3` the entry would be:

```
example( pos, tool_pose( stick3, s7)).
```

The primitive world state corresponding to this example state is also recorded in

Algorithm 9 Learn the *tool_pose* predicate by trial-and-error.

LEARN_BY_TRIAL_AND_ERROR

```
1: Read the user-defined task file
2: Set learning target as tool_pose(Tool,State)
3: Generate new instance of learning task
4: plan ← Generate a new plan
5: repeat
6:    tool ← SELECT_NEW_TOOL
7:    Insert tool into plan
8:    Execute plan, including novel tool action
9:    learnState ← world state encountered at start of novel action
10:   if plan succeeded then
11:      add learnState as positive example of tool_pose(Tool,State)
12:      h_s ← revise most specific hypothesis (ILP)
13:      Generate new learning task instance
14:   else
15:      Add learnState as negative example of tool_pose(Tool,State)
16:      h_g ← revise most general hypothesis (ILP)
17:      Reset current learning task
18:   end if
19: until max_trials_completed
```

an accompanying file.

Having recorded a new learning example, the agent revises its most-specific (h_s) or most-general (h_g) hypotheses describing the tool pose state. These hypotheses affect the future actions of the agent in two ways: they determine how a new tool is selected (the tool structure), and they determine how the tool is used (the tool pose).

Tool selection is governed by the SELECT_NEW_TOOL procedure shown in Algorithm 10. This algorithm examines each potential tool in turn and scores it according to its structural similarity to the current most-specific and most-general hypotheses. Recall from Chapter 4 that a structural literal is defined

Algorithm 10 Select a new tool.

SELECT_NEW_TOOL

1: $h_g,\ h_s \leftarrow$ current hypothesis
2: **for** each *obj* in available tools **do**
3: $\quad score \leftarrow$ SCORE_STRUCTURAL_SIMILARITY(h_g, h_s, *obj*)
4: **end for**
5: *bestTool* $\leftarrow$ tool with highest score
6: **return** *bestTool*

SCORE_STRUCTURAL_SIMILARITY (h_g, h_s, *tool*)

1: $h'_s \leftarrow$ structural literals in h_s
2: $h'_g \leftarrow$ structural literals in h_g
3: Set tool parameter in h'_g and h'_s equal to *tool*
4: *score* $\leftarrow$ number of satisfiable literals in h'_s
5: **if** any literals in h'_g are not satisfiable **then**
6: $\quad score \leftarrow 0$
7: **end if**
8: **return** *score*

as one which is time-independent (ie. contains no "state" parameter), such as `length(Tool,Length)`. An object receives a score equal to the number of structural literals in h_s which it satisfies. However, any object which does not satisfy all of the structural literals in h_g receives a score of zero. The SELECT_NEW_TOOL algorithm then chooses the tool with the highest score. As the learner's current definition of h_g and h_s changes so too does the score assigned to any particular object.

Tool pose generation is likewise affected by both h_g and h_s, although the algorithm is more complex and involves the use of the constraint solver module described in Section 6.3.4. Tool pose generation occurs each time the agent attempts to execute an action, and in the case of novel actions a special version of the algorithm is used. This algorithm was detailed in Chapter 4 (see Algorithm 1) and so will not be discussed further here.

Generation of new hypotheses

The agent's ILP algorithms are invoked after each new learning example has been collected, in order to revise the current hypothesis describing the tool pose state. As described in Chapter 4, our algorithms for generating revised hypotheses from examples are based on Muggleton's GOLEM algorithm (Muggleton and Feng, 1992). Here we give some further detail on how they are implemented.

The most-specific hypothesis h_s is generated by firstly selecting pairs of positive examples and calculating their least-general generalisation (lgg). The pair whose lgg has the best coverage provides the starting point for the search. The algorithm then repeatedly tries to improve its hypothesis by including additional positive examples in the lgg. This process continues until coverage stops improving or all of the examples are covered. The pseudo-code for generating the most-specific hypothesis is given in Algorithm 2 (Chapter 4). Our implementation is once again written in SWI-Prolog.

The key step in generating a new most-specific hypothesis is the computation of the (relative) least-general generalisation of two or more examples. The rules for generating our version of the lgg (a mode-restricted lgg) were given in 4.3.5, and the corresponding pseudo-code is presented here in Algorithms 11 through 13.

The first step in the lgg algorithm is to create a new clause representing each example. A clause is created from an example by setting `tool_pose` as the head of the clause and saturating the body with literals listed in the agent's mode declarations (supplied as background knowledge). This results in a clause:

```
tool_pose( Tool, State) :- a1, a2, ..., an
```

where the `ai` are the generated body literals, and `State` and `Tool` are given by the example. The saturation process we have used works in the usual manner, by chaining forward from the head of the clause and using the mode declarations to generate all valid body literals up to a given depth. A body literal is generated by instantiating its input parameters with terms appearing as output parameters earlier in the clause (or input parameters of the clause head).

The lgg of the two clauses representing the examples can then be calculated. The pseudo-code for this process is given in Algorithm 11. It works by calculating

Algorithm 11 Compute the lgg of positive examples.

LGG(*exampleA*, *exampleB*)

1: *clauseA* ← saturate *exampleA* clause with background knowledge
2: *clauseB* ← saturate *exampleB* clause with background knowledge
3: Let *headA*, *bodyA* be the head and body of *clauseA*
4: Let *headB*, *bodyB* be the head and body of *clauseB*
5: *lgghead*, *subs* ← LGG_LIT(*headA*, *headB*)
6: **for** i = 1 to maxdepth **do**
7: *lgg*, *newsubs* ← LGG_BODY(*bodyA*, *bodyB*, *subs*)
8: *lggbody* ← *lggbody* ∪ *lgg*
9: *subs* ← *subs* + *newsubs*
10: **end for**
11: return (*lgghead* :– *lggbody*)

LGG_BODY(*bodyA*, *bodyB*, *subs*)

1: **for** each literal a_i in *bodyA* **do**
2: **for** each literal b_j in *bodyB* **do**
3: lgg_{ij}, $newsubs_{ij}$ ← LGG_LIT(a_i, b_j, *subs*)
4: **if** lgg_{ij} = none **then**
5: **loop**
6: **else**
7: *lgg* ← *lgg* + lgg_{ij}
8: *newsubs* ← *newsubs* ∪ $newsubs_{ij}$
9: **end if**
10: **end for**
11: **end for**
12: return (*lgg*, *newsubs*)

the lgg of the two clause heads, and then finding the lgg of each pair of body literals from either clause. The lgg of a pair of literals depends on the available substitutions, and so this process of generating body literal lggs is repeated up to a user-defined 'depth'. During each loop zero or more new variable substitutions may be introduced.

Algorithm 12 Compute the mode-restricted lgg of two literals.

LGG_LIT(*lita*, *litb*, *subs*)

1: **if** functor(*lita*) $\neq$ functor(*litb*) **or** arity(*lita*) $\neq$ arity(*litb*) **then**
2: **return** (none, none)
3: **end if**
4: Let $a_1, a_2, \ldots, a_n$ be the parameters of *lita*
5: Let $b_1, b_2, \ldots, b_n$ be the parameters of *litb*
6: Let $m_1, m_2, \ldots, m_n$ be the modes of $a_1, a_2, \ldots, a_n$
7: Let $n_1, n_2, \ldots, n_n$ be the modes of $b_1, b_2, \ldots, b_n$
8: **for** i = 1 to n **do**
9: lgg_i, newsub$_i$ $\leftarrow$ LGG_TERM(a_i, b_i, m_i, n_i, *subs*)
10: **if** lgg_i = none **then**
11: **return** (none, none)
12: **end if**
13: *newsubs* $\leftarrow$ *newsubs* + *newsub*$_i$
14: **end for**
15: define *lgg* with
16: - same functor as *lita* and *litb*
17: - parameters equal to lgg_i
18: **return** (*lgg*, *newsubs*)

The algorithm for determining the lgg of a pair of body literals is shown as LGG_LIT in Algorithm 12. It involves finding the lgg of each pair of parameters, using the function LGG_TERM (Algorithm 13). In order for the lgg of a pair of literals to be defined, the lgg of each parameter pair must be defined.

The crucial point in our mode-restricted lgg is that the lgg of two terms (see LGG_TERM) depends upon the mode of the parameter and the set of valid substitutions which already exist in the lgg. Given a pair of ground terms x and y, a new variable X, may only be introduced if x and y are output parameters and no Y: x/y substitution already exists for these parameters. In addition, this new variable will only survive if the lgg of all of the input parameters pairs in the literal are well-defined.

Algorithm 13 Compute the mode-restricted lgg of a pair of terms.

```
LGG_TERM( terma, termb, modea, modeb, subs)
 1: if modea≠modeb then
 2:    lgg ← none
 3:    newsub ← none
 4: else if terma = termb  then
 5:    lgg ← terma
 6:    newsub ← none
 7: else if if modea = '−' then
 8:    if X:terma/termb ∈ subs  then
 9:       lgg ← X
10:       newsub ← none
11:    else
12:       create new substitution Y:terma/termb
13:       lgg ← Y
14:       newsub ← Y:terma/termb
15:    end if
16: else if modea = '+'  then
17:    if X:terma/termb ∈ subs then
18:       lgg ← X
19:       newsub ← none
20:    else
21:       lgg ← none
22:       newsub ← none
23:    end if
24: end if
25: return (lgg, newsub)
```

The most-specific hypothesis h_s is constructed as the lgg of two or more positive examples, as described above, including new variables which may not exist in the head of the clause. The most-general hypothesis h_g describing the tool pose state may then be generated directly from h_s via the process of negative-based reduction. Negative-based reduction attempts to find a shorter (more general)

clause by deleting literals from the hypothesis in a greedy manner. The pseudo-code for this algorithm, which appeared as part of GOLEM, is detailed at the end of Chapter 4.

Chapter 7

Conclusions and future work

In this thesis we have presented a robot agent which is able to learn to use objects as tools to solve problems. The agent employs both explanation-based and inductive approaches to learn the correct way in which a tool should be used, what the tool is useful for, and what the key properties are that make an object a suitable tool. Our approach involves learning at the abstract relational level, which allows our agent to generalise over objects and situations more broadly than previous robot tool-learning approaches.

This chapter summarises the research presented in this thesis and outlines the most interesting and challenging ways in which the work could be extended.

7.1 Summary

The main goal of this research was to build a robot agent capable of learning to solve problems by using objects as tools. We introduced a broad definition of a tool as any object which enables the agent to achieve a goal which would have otherwise been more difficult or impossible to achieve.

One of the distinguishing features of our approach is that our agent is learning tool-use in the context of problem-solving. The agent's learning is motivated by the need to solve a problem, and it is able to identify the purpose and benefits of using a tool by watching another agent — rather than having tool subgoals defined a priori by a programmer.

The other main difference in our approach is that we have emphasised relational learning, and an ability to generalise over situations and tool objects. An ability to generalise effectively is essential for a general-purpose robot because it never has to solve exactly the same problem twice. Previous work on tool-use learning in robots has focused at the more primitive level of learning manipulation behaviours.

The main contributions of this thesis are:

- The integration of tool-use learning and problem solving in a robot agent.
- The use of a relational learner for solving robot tool-use learning problems. Our agent can learn important tool-use concepts which cannot be represented by the approaches used in previous work.
- A novel action representation which integrates symbolic planning, constraint solving, and motion planning. We use STRIPS models and first-order relations to constrain motion planning behaviours.
- A novel method for learning action models via a form of explanation-based learning.
- A novel method for incremental learning of relational concepts, based upon a version-space-like representation and a restricted form of least general generalisation.

7.2 Lessons and limitations

A number of general conclusions can be drawn from the work we have completed on tool-use learning. The lessons learned here may be useful to researchers contemplating an extension of our work, or to those who are working in a related area.

Learning bias

A key characteristic of our approach was the learning bias employed, involving the use of both the most-specific and most-general forms of the hypothesis. This contrasts strongly with the majority of existing work in ILP in which either a purely top-down or bottom-up approach is adopted. Each of these approaches,

used in isolation, would have suffered from significant disadvantages in our tool-use domain.

The top-down (general-to-specific) approach has the disadvantage that it learns a purely *discriminative* hypothesis — the shortest possible description of how a positive example can be distinguished from a negative one. In the case where the agent has only a few examples to work with (as in our experiments) this clause is typically only one or two literals in length. Short hypotheses such as these provide only limited information about how an action should be executed. In order to obtain a more detailed hypothesis it is necessary to collect a very large number of learning examples.

In contrast, the bottom-up (specific-to-general) approach very quickly arrives at a detailed description of the action, a so-called *characteristic* hypothesis. This hypothesis contains all of the essential elements which the observed positive examples have in common. Unfortunately, the most-specific hypothesis only contains information about positive examples and is usually so overly-specific that it is not directly applicable to the current world situation.

Our approach resolves these problems by making use of both most-specific and most-general hypotheses. This allows the hypothesis space to explored more efficiently, as the agent can ensure that it selects new examples to test which lie somewhere in between these two hypothesis boundaries. It should be noted that this advantage is most useful in the incremental learning scenario, such as the learning problems addressed in this thesis.

Bootstrapping inductive learning with explanation-based learning

One of the useful lessons which has come out of this research is that explanation-based learning (EBL) can play an effective role in "bootstrapping" trial-and-error learning of action models.

As we discussed in Chapter 2 (section 2.2.2), EBL has an unfortunate reputation of being unable to learn anything which was not already implied by the background knowledge. This reputation has no doubt come about because of EBL's origins in speedup learning and learning action "macros". However, in the case where the agent's domain theory is *incomplete* it is indeed possible for EBL to

introduce new domain knowledge. This is of course the exact situation addressed in this thesis, where an explanation allows the agent to infer useful information about a novel action.

The advantage of the explanation-based approach is that it is able to learn a significant amount of information from a single example. This is important in an action-learning context because it allows an agent to quickly focus its experimentation on a narrower part of the hypothesis space. In this manner, explanation-based learning provides an effective starting point for inductive trial-and-error learning and allows the agent to learn from fewer examples.

One of the goals of our work was to present a learning system which would be able to define its own learning objectives, motivated by the desire to achieve a goal. We would argue that all tool-use is goal-oriented, and so it makes sense to learn tool-use in a goal-oriented context. Explanation-based learning was an essential component of making this possible, by providing the agent the ability to recognise and explain novel behaviours which it might wish to learn.

Close interaction between geometric and logical representations of space

Solving tool-use problems involves an interesting mix of abstract logic and low-level object interaction. To reflect this fact, a key aspect of our approach is the very close interaction between logical spatial predicates and a primitive geometric model of the world. This interaction is manifested in both action execution and action learning.

In our approach to action representation and execution, the spatial goal predicates in STRIPS models are transformed to motion plans in geometric space. This process is facilitated through the use of a constraint solver, which is able to find geometric solution poses satisfying a set of logical constraints. This tight integration between "geometric" (motion) planning and abstract (STRIPS) planning allows our agent to solve problems at both the abstract and fine-grained manipulation level. Our review of the literature in Chapter 2 notes a number of related pieces of work which have adopted the same theme.

The close interplay between geometric and logical models of objects and space is also apparent in our approach to action learning. Our learning-by-explanation al-

gorithm relies on low-level motion and contact-based segmentation of the teacher's example, before a more abstract explanation is constructed using STRIPS models. The learning by trial-and-error algorithm also makes heavy usage of a geometric model in testing hypotheses, as discussed in Section 5.5. The collision checker and constraint solver allow the agent to eliminate non-physical solution poses, without needing to test them physically.

Whilst our approach has been developed to specifically address tool-use learning problems, it seems likely that some of the ideas developed in our work could be useful more generally — particularly in other spatial domains.

Determinacy assumption

As noted in Chapter 4, the ILP component of our learning algorithm which deals with generating a new most-specific hypothesis is based on GOLEM. One of the significant limitations of GOLEM which also applies to our work is the assumption of determinacy in the background predicates. The determinacy assumption states that the values of any output parameters of a predicate should be determined uniquely by the value of its input parameters. The determinacy restriction ensures that the hypothesis clause grows at worst polynomially with the number of positive examples.

In practice the determinacy restriction can make it awkward to describe some physical objects. Consider, for example, describing a table with four legs. It would seem natural to describe the fact that each leg is attached to the table top by assertions such as `attached( tabletop, leg1)` and `attached( tabletop, leg2)` and so on. However if the second parameter of `attached` is an output parameter this would violate the determinacy restriction.

Determinacy could be imposed artificially in this example by writing the attachment predicate as four separate predicates (`attached1`, `attached2` ...), representing the "first" attachment, "second" attachment and so on. This approach will only be useful in cases where each of the attachments are identical (eg. in the case of table legs) or we can apply a heuristic ordering to the attachments. In other cases, an alternative modification to the lgg algorithm would need to be developed to limit the number of valid lggs.

Assumption of complete action models

Another important limitation of our approach is the assumption that the action models supplied to the agent are accurate. In reality, an agent would be continually refining each of its action models as it gains experience — as a consequence its current set of models would contain errors or omissions. Our approach to learning by explanation assumes that the agent's existing models are correct, which allows it to confidently spot novel actions during the course of the teacher's example. If we allow for the fact that agent's action models may contain significant errors then the question arises — does a gap in the explanation represent a novel action, or simply a unrecognised action due to an incorrect action model?

Fortunately it seems likely that our approach could be suitably modified to take account of the possibility of errors in the agent's action models. For example, an agent might be able to explain a 'gap' by making a minor revision to one of its existing action models — the proposed correction could then be tested through experimentation.

Noise-free assumption

The final limitation we will remind the reader of here is that our approach assumes that the state signal from the world is noise-free. Introducing significant noise would certainly complicate the learning-by-explanation algorithm, since it would be difficult to reliably distinguish from a single teacher example whether an "interesting" effect was the result of a novel action or simply noise. The most obvious solution to this problem would seem to be to allow the agent access to more than one demonstration example from the teacher. The learning-by-explanation problem would then involve an inductive component to help isolate noise in the example.

7.3 Future work

The most interesting extensions to this work would involve extending the world representation and learning algorithms to allow a greater range of tool problems

to be tackled. These include learning skillful manipulation of tools, analogical problem-solving for tool-use problems, learning repetitive tool actions, and the representation and use of non-rigid body tools and objects.

A more challenging task would be to implement the existing learning system on a robot operating in the real world. This would involve addressing considerable challenges in sensing and control which we have not addressed in this thesis. It would, however, allow the agent to tackle some interesting real world tool-use problems which are difficult to simulate (such as mopping up a liquid).

7.3.1 Learning complex manipulation behaviours

There is considerable scope in future work for more sophisticated learning of object and tool manipulation at the primitive level.

Due to the complex nature of the object manipulation problem, in our research we have deliberately chosen to examine tool use problems where dexterous manipulation is not required. We have instead focused on learning how tools can be incorporated into planning, and on learning the relations which must exist between tools, objects, the user and the surrounding environment. It is a fact however that many tools can be used more effectively, usefully and easily when manipulated by a skillful user.

As an example we might wish to consider the problem of an agent learning to hit a nail into a piece of wood with a hammer. Our current system would be sufficient for learning the necessary properties of the hammer, along with the necessary spatial relationships which must exist between the hammer head and the nail. However, it would be unable to replicate the proper hitting behaviour demonstrated by the teacher. Instead it would try to plan a path for the hammer head, in order to move the head into contact with the nail; it would then attempt to push the nail into the wood with the hammer head. A better system would learn as a subgoal the desired velocity of the hammer head at the point of contact, not just the relative pose. The motion planner would therefore need to operate in velocity space as well as pose space.

A more sophisticated approach might try to marry our abstract model learning approach with a primitive behaviour learner at the lower level. As in the work

presented here, the action model would define the desired goal of the lower level behaviour. However the lower level behaviour would then be learnt rather than implemented by a motion planner. A similar idea has been addressed in previous work by Ryan (2004), where the agent is provided with an abstract action model by the user and must learn a behaviour (via reinforcement learning) which implements it.

7.3.2 Richer action models

The action models used in our current implementation of the architecture are based on a simple extension to the parameterised STRIPS model (see Chapter 3). Whilst these basic action models were sufficient for the problems addressed in this research, it would be interesting to incorporate richer action models into the system. This would allow us to address problems in which conditional effects, stochastic actions and concurrent execution are required, for example. More generally, a worthwhile project would be to extend the architecture to support action models defined by the PDDL (Planning Domain Definition Language) standard (McDermott, 2000).

In order to make these extensions changes would need to be made to the learning by explanation, learning by trial-and-error, and planning modules. The planning module would be replaced by a suitable PDDL planner according to the requirements of the problem or preferences of the user[1]. The learning by explanation module would require straightforward modification to account for actions with conditional or stochastic effects and concurrent execution, but the fundamental algorithm would remain largely unchanged.

The trial-and-error module would present the greatest challenge to the implementation of richer action models — at least in the case where the novel action to be learnt is an action with conditional or stochastic effects. Our current approach to learning the new action model assumes that there is a single set of effects which occur, and can be observed from the teacher's example. In order to extend this it would be necessary to adopt a more complex approach to learning action effects,

[1]Although it should be noted that our current Fast-Forward planner can handle a number of PDDL extensions such as conditional effects.

such as that presented in Pasula et al. (2007).

7.3.3 Repetitive tool actions

Many tool actions involve repetitive application of the tool in order to achieve the desired goal — peeling a carrot, sweeping a floor, or painting a wall for example. Incorporating these types of actions into our learner would involve modifying the learning from explanation module to recognise repeated subsequences of actions, where the overall goal of the action is not achieved until a number of repetitions of the tool-object interaction have been carried out.

Repetitive tool actions would also require modifications to the way actions are represented and executed in our architecture. Repetitive actions often involve moving across a defined region or surface, where each repeat of the action sequence "covers" part of the goal region. The need to cover an area with multiple repetitions of an action could perhaps be incorporated into an extended version of a motion planner.

7.3.4 Analogical problem solving

Some tool-use problems are related on an abstract conceptual level. For example, using a ladder to reach a light bulb and using a stick to reach an object on a shelf are conceptually similar problems. Our existing agent relies on a demonstration from a teacher agent to seed each learning task. In some situations it should be possible to avoid this step by using analogical reasoning. For example, if the agent already has an abstract action for box-pushing it could hypothesise an equivalent action involving the use of a tool which would act on the object instead.

7.3.5 Implementation on a real-world robot

The next logical step in the development of our tool learning system would be to apply it to a robot operating in the real world. Doing so would involve overcoming a number of significant challenges in sensing and actuation, as we discuss below.

The robot platform used in our simulator, the Pioneer 2 robot, imposed significant constraints on the tool-learning experiments we were able to carry out.

Although the Pioneer platform was sufficient to demonstrate the tool-learning abilities of our system, somewhat ironically the "simplicity" of the platform made life more difficult than it would have been with a more complex, but flexible robot platform.

The major limitation was imposed, not unexpectedly, by the limited object manipulation abilities of the Pioneer robot. The simple form of the Pioneer gripper meant that using tools in a vertical plane of motion was difficult or impossible. This greatly restricted the variety of tool-use problems that our system was able to attempt.

In addition, the differential constraints imposed by the non-holonomic nature of the Pioneer robot meant that manipulation of tools and objects was more difficult than it would have been with a robot capable of making omni-directional movements. The Pioneer cannot move sideways so it is difficult to correct small lateral errors in desired pose without rotating the robot by 90 degrees in order to shift its position 'sideways'. This makes both path-planning and path-following a more difficult problem for a differentially constrained robot. These problems are compounded when the robot is holding a lengthy tool, since small errors in robot pose can cause large errors in the pose of the end of the tool, and small adjustments are not possible without making a series of maneuvers (such as reversing, turning slightly, and then moving back in). An omni-directional robot would be able to make small adjustments much more simply.

Implementation on a robot arm

With these points in mind, the most straightforward real-world robot platform would be to use a multi-jointed arm mounted on a fixed platform as the robot, with a camera mounted above the base of the arm. This setup is commonly used in robotics system research, and was in fact used in Stoytchev's work on a developmental approach to robot tool use (Stoytchev, 2007)). A suitable robot arm was not available in our lab at the time this research was commenced — hence the choice of the Pioneer as our robot platform in this research.

There are a number of advantages to using a robot-arm on a fixed platform (with an overhead camera) to implement a real-world tool learning system. For

the vision system, the benefits of the fixed overhead camera include:

- no object pose uncertainty due to robot/camera motion
- less occlusion of objects than would occur with a camera mounted lower
- no changes in lighting due to changes in viewpoint

The greater flexibility of movement in the arm/gripper combination would also make things easier for object manipulation. Advantages include:

- more object affordances are available for grasping
- objects can be moved into any orientation
- omni-directional movement possible, so small corrections to object poses are easy to make

In short, a robot-arm based system would make both the sensing and object control tasks simpler, meaning that a wider and more interesting range of tool-use problems could be studied. A simulated version of the arm could still be used for development and testing, although only for problems involving rigid body objects.

One of the benefits of moving to a real-world robot implementation would be the ability to learn tool-use problems which are difficult or impossible to study in a rigid-body simulator. Some examples of the type of tool-use problems which would be interesting to investigate include:

- mopping up liquid with a sponge
- using a knife to cut with
- peeling a carrot with a peeler
- buttering a piece of toast with a knife

Key implementation issues

A significant difficulty in a real-world implementation of our work would be to obtain sufficiently robust sensing of object poses. In our simulation we were able to ignore this issue by obtaining a global object pose directly from the simulator. In a real-world scenario however this is obviously not feasible and the robot must estimate object poses (or relative object poses) only from its noisy sensor data. A

huge amount of work would need to be done to get to a stage where the noise in the state signal was at a manageable level.

Furthermore, a robot's actions are rarely deterministic, and an executed action does not always produce the desired effects. Learning in the presence of a noisy state signal and non-deterministic actions is a significant challenge which would require aspects of our system to be considerably modified.

Most of these changes in our learning algorithms would need to occur in the learning from explanation module. This module currently assumes a noise-free trace of the demonstration in order to construct a correct explanation of the teacher's actions. This is due to the fact that we are learning from a single example and it would be very difficult to distinguish between noise and a novel action in this case. One of the solutions to this problem would be to supply the agent with multiple demonstration examples. The explanation-based learner would then be in a better position to be able to distinguish noise from the actual actions executed by the teacher.

Our relational tool-learning algorithm in the trial-and-error learning module of our agent is already able to handle noise in the examples it receives, so fewer changes would be needed here. However, more examples would be required in order to learn a concept of the same length.

The implementation of our approach on a robotic arm would also involve addressing the challenge of working with a system with many more degrees of freedom. At the level of the motion planner, the RRT planner we have used is ideally suited to problems with high dimensionality such as motion-planning for humanoid robots (Chestnutt et al., 2005) and this module would therefore require only straightforward modification.

Representing a multi-jointed robotic arm in abstract logic, although not difficult, would complicate the action learning (ILP) process by introducing additional objects into the world state. The hypothesis space would then grow to include additional relations between components of the arm and the objects the agent or tool is interacting with. It would seem reasonable in many instances to hide this extra complexity from the ILP learner by ignoring the extra arm components, at least initially. After all, in most practical tool-use situations it is the relative po-

sition of the tool and gripper which is most important. The implications of the complex geometry of the arm would then simply be captured at the level of the motion planner.

An interesting extension of our work along these lines would be an iterative-deepening style approach to identifying "relevant" objects from which the hypothesis is constructed. The algorithm would start by considering the interactions between the most important objects, such as the tool and the object it is interacting with. If a suitable hypothesis is not found the search would expand to include relations between other nearby objects, or components of the robot arm. A distance heuristic would be a simple way of determining the order in which new objects are added into consideration.

Bibliography

Abbeel, P., Coates, A., Quigley, M., and Ng, A. (2007). An application of reinforcement learning to aerobatic helicopter flight. In *Proceedings of the Advances in Neural Information Processing*.

Abbeel, P. and Ng, A. (2004). Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the 21st International Conference on Machine Learning*.

Amir, E. (2005). Learning partially observable deterministic action models. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 1433–1439, Edinburgh, Scotland, UK.

Apt, K. R. and Wallace, M. G. (2007). *Constraint Logic Programming using ECLiPSe*. Cambridge University Press.

Argall, B., Chernova, S., Veloso, M., and Browning, B. (2009). A survey of robot learning from demonstration. *Robotics And Autonomous Systems (to appear)*, doi:10.1016/j.robot.2008.10.024.

Atkeson, C. and Schaal, S. (1997). Robot learning from demonstration. In *Proceedings of the 14th International Conference on Machine Learning*, pages 12–20.

Baber, C. (2003). *Cognition and tool use: Forms of engagement in human and animal use of tools*. Taylor & Francis.

Beck, B. B. (1980). *Animal tool behaviour: The use and manufacture of tools by animals*. Taylor & Francis.

Benson, S. (1996). *Learning action models for reactive autonomous agents*. PhD thesis, Department of Computer Science, Stanford University.

Bogoni, L. (1995). *Identification of functional features through observations and interactions*. PhD thesis, University of Pennsylvania.

Bratko, I., Urbancic, T., and Sammut, C. (1998). Behavioural cloning: Phenomena, results and problems. In *IFAC Symposium*, Berlin.

Brown, S. and Sammut, C. (2007). An architecture for tool use and learning in robots. In *Proceedings of the 2007 Australasian Conference on Robotics and Automation*.

Cambon, S., Alami, R., and Gravot, F. (2009). A hybrid approach to intricate motion, manipulation and task planning. *International Journal of Robotics Research*, 28(1):104–126.

Cambon, S., Gravot, F., and Alami, R. (2004). aSyMov: Towards more realistic robot plans. In *International Conference on Automatic Planning and Scheduling*.

Chappell, J. and Kacelnik, A. (2002). Tool selectivity in a non-primate, the New Caledonian crow (Corvus moneduloides). *Animal Cognition*, 5:71–78.

Chappell, J. and Kacelnik, A. (2004). Selection of tool diameter by New Caledonian crows Corvus moneduloides. *Animal Cognition*, 7:121–127.

Chestnutt, J., Lau, M., Kuffner, J., Cheung, G., Hodgins, J., and Kanade, T. (2005). Footstep planning for the ASIMO humanoid robot. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2005)*.

Christiansen, A., Mason, M., and Mitchell, T. (1990). Learning reliable manipulation strategies without initial physical models. In *Proceedings of the 1990 IEEE International Conference on Robotics and Automation*, pages 1224–1230, Cincinnati, Ohio.

Cruse, H. (2003). The evolution of cognition: A hypothesis. *Cognitive Science*, 27:135–155.

Dietterich, T. (1990). Exploratory research in machine learning. *Machine Learning*, 5(1):5–9.

Ehmann, S. (2001). SWIFT – A library for collision detection, distance computation, and contact determination. http://gamma.cs.unc.edu/SWIFT/.

Ehmann, S. and Lin, M. (2000). SWIFT: Accelerated Proximity Queries Between Convex Polyhedra By Multi-Level Voronoi Marching. In *Proceedings of the International Conference on Intelligent Robots and Systems*.

Fikes, R. and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208.

Fitzpatrick, P., Metta, G., Natale, L., Rao, S., and Sandini, G. (2003). Learning about objects through action — initial steps toward artificial cognition. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 03)*, volume 3, pages 3140–3145.

Fuentes, O. and Nelson, R. (1998). Learning dextrous manipulation skills for multifingered robot hands using the evolution strategy. *Machine Learning*, 31:223–237.

Gat, E., Bonnasso, R., and Murphy, R. (1998). On three-layer architectures. In *Artificial Intelligence and Mobile Robots*, pages 195–210. AAAI Press.

Gerkey, B. P., Vaughan, R. T., and Howard, A. (2003). The Player/Stage Project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, Coimbra, Portugal.

Gil, Y. (1993). Efficient domain-independent experimentation. In *Proceedings of the Tenth International Conference on Machine Learning*.

Gil, Y. (1994). Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of the Eleventh International Conference on Machine Learning*.

Goodall, J. and van Lawick, H. (1966). Use of tools by egyptian vultures. *Nature*, 212:1468–1469.

Guitton, J. and Farges, J.-L. (2008). Geometric and symbolic reasoning for mobile robotics. In *Proceedings of the 3rd National Conference on Control Architectures of Robots*, Bourges, France.

Hirano, Y., Kitahama, K., and Yoshizawa, S. (2005). Image-based object recognition and dexterous hand/arm motion planning using RRTs for grasping in cluttered scene. In *Proceedings of IEEE/RSJ Intl.\ConferenceonIntelligentRobotsandSystems(IROS*2005).

Hoffmann, J. and Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:2001.

Hume, D. (1995). *Induction of procedures in simulated worlds*. PhD thesis, University of New South Wales.

Hunt, G. (1996). Manufacture and use of hook-tools by New Caledonian crows. *Nature*, 379:249–251.

Isaac, A. and Sammut, C. (2003). Goal-directed learning to fly. In *Proceedings of the 20th International Conference on Machine Learning*, pages 258–265.

Kambhampati, S. and Yoon, S. (2008). *Encyclopedia of Machine Learning*, chapter Explanation-based learning for planning. Springer-Verlag, New York.

Katz, D., Pyuro, Y., and Brock, O. (2008). Learning to manipulate articulated objects in unstructured environments using a grounded relational representation. In *Proceedings of Robotics: Science and Systems IV*.

Kemp, C. C. and Edsinger, A. (2006). Robot manipulation of human tools: Autonomous detection and control of task relevant features. In *Proceedings of the Fifth International Conference on Development and Learning, Special Session on Classifying Activities in Manual Tasks*.

Kenward, B., Weir, A., Rutz, C., and Kacelnik, A. (2005). Tool manufacture by naive juvenile crows. *Nature*, 433(121).

Koenig, N. and Howard, A. (2004). Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS 2004)*, volume 3, pages 2149–2154.

Kohler, W. (1925). *The Mentality of Apes*. Harcourt, Brace & Co, New York. Translated from German by Winter, E.

Krützen, M., Mann, J., Heithaus, M. R., Connor, R. C., Bejder, L., and Sherwin, W. B. (2005). Cultural transmission of tool use in bottlenose dolphins. *Proceedings of the National Academy of Sciences*, 102(25):8939–8943.

Kuffner, J. and LaValle, S. (2000). RRT-Connect: An efficient approach to single-query path planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation (ICRA'2000), San Francisco, CA*.

Kuffner, J., Nishiwaki, K., Kagami, S., Inaba, M., and Inoue, H. (2003). Motion Planning for Humanoid Robots. In *Proceedings of the 11th Intl.\Symp.\of RoboticsResearch(ISRR*2003).

Kuniyoshi, Y., Inaba, M., and Inoue, H. (1994). Learning by watching: Extracting reusable task knowledge from visual observation of human performance. *IEEE Transactions on Robotics and Automation*, 10(6):799–822.

LaValle, S. and Kuffner, J. (1999). Randomized kinodynamic planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Detroit, MI.

Lavrac, N. and Dzeroski, S. (1994). *Inductive logic programming: Techniques and applications*. Ellis Horwood.

Levey, D., Duncan, R., and Levins, C. (2004). Use of dung as a tool by burrowing owls. *Nature*, 431:39.

Levine, G. and DeJong, G. (2006). Explanation-based acquisition of planning operators. In *ICAPS*, pages 152–161.

Lorenzo, D. and Otero, R. (2000). Learning to reason about actions. In *Proceedings of the 14th European Conference on Artificial Intelligence*.

Lynch, K. (1993). Estimating the friction parameters of pushed objects. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 186–193.

Mason, M., Christiansen, A. D., and Mitchell, T. (1989). Experiments in robot learning. In *Proceedings of the Sixth International Workshop on Machine Learning*. Morgan Kaufmann.

McCarthy, J. and Hayes, P. (1969). Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502.

McDermott, D. (2000). The 1998 AI planning systems competition. *AI Magazine*, 21(2).

Michie, D. (1993). Knowledge, learning and machine intelligence. *Intelligent Systems*, pages 2–19.

Mitchell, T. (1978). *Version space: An approach to concept learning*. PhD thesis, Stanford University.

Mitchell, T. (1982). Generalization as search. *Artificial Intelligence*, 18:203–266.

Mitchell, T., Keller, R., and Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80.

Morales, E. and Sammut, C. (2004). Learning to fly by combining reinforcement learning with behavioural cloning. In *Proceedings of the 21st International Conference on Machine Learning*.

Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286.

Muggleton, S. and Feng, C. (1992). Efficient induction of logic programs. In Muggleton, S., editor, *Inductive Logic Programming*, pages 281–298. Academic Press.

Narasimhan, S. (1995). *Task-level strategies for robot tasks*. PhD thesis, Deparment of Computer Science and Electrical Engineering, Massachusetts Institute of Technology.

Ng, A. and Russell, S. (2000). Algorithms for inverse reinforcement learning. In *Proceedings of the International Conference on Machine Learning*.

Nicolescu, M. and Mataric, M. (2001). Learning and interacting in human-robot domains. *IEEE Transactions on Systems, Man., and Cybernetics — Part A: Systems and Humans*, 31(5):419–430.

Nicolescu, M. and Mataric, M. (2003). Natural methods for robot task learning: Instructive demonstrations, generalization and practice. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 241–248.

Nilsson, N. J. (1984). Shakey the Robot. Technical note 323, SRI International, Menlo Park, CA.

Oates, T. and Cohen, P. (1996). Searching for planning operators with context-dependent and probalistic effects. In *Proceedings of the Thirteenth National Conference On Artificial Intelligence*, pages 865–868. AAAI Press.

Palhang, M. and Sowmya, A. (1997). *Visual Information Systems*, chapter Automatic acquisition of object models by relational learning, pages 239–258. Lecture Notes in Computer Science. Springer Berlin.

Pasula, H., Zettlemoyer, L., and Kaelbling, L. (2004). Learning probabilistic planning rules. *International Conference on Automated Planning and Scheduling*.

Pasula, H. M., Zettlemoyer, L. S., and Kaelbling, L. P. (2007). Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352.

Plotkin, G. (1970). A note on inductive generalization. In Meltzer, B. and Mitchie, D., editors, *Machine Intelligence*.

Pollard, N. and Hodgins, J. (2002). Generalizing demonstrated manipulation tasks. In *Workshop on the Algorithmic Foundations of Robotics (WAFR 02)*, Nice, France.

Potts, D. (2007). *Learning to control*. PhD thesis, University of New South Wales.

Povinelli, D. (2000). *Folk physics for apes: the chimpanzee's theory of how the world works*. Oxford University Press, Oxford.

Rao, C., Yilmaz, A., and Shah, M. (2002). View-invariant representation and recognition of actions. *International Journal of Computer Vision*, 50(2):203–226.

Ratliff, N., Bagnell, J., and Zinkevich, M. (2006). Maximum margin planning. In *International Conference on Machine Learning*.

Ryan, M. R. (2004). *Hierarchical reinforcement learning: A hybrid approach*. PhD thesis, School of Computer Science and Engineering, University of NSW.

Salganicoff, M., Metta, G., Oddera, A., and Sandini, G. (1993). A vison-based learning method for pushing manipulation. In *AAAI Fall Symposium Series: Machine Learning in Vision: What Why and How?*, Raleigh, N.C.

Salganicoff, M., Ungar, L., and Bajcsy, R. (1996). Active learning for vision-based robot learning. *Machine Learning*, 23:251–278.

Sammut, C., Hurst, S., Kedzier, D., and Michie, D. (1992). Learning to fly. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 385–393.

Schmill, M., Schmill, D., Oates, T., and Cohen, P. (2000). Learning planning operators in real-world, partially observable environments. In *Proceedings of the Fifth International Conference on Artificial Intelligence, Planning and Scheduling*.

Shen, W. (1994). *Autonomous Learning from the Environment*. Computer Science Press, W.H. Freeman and Company.

Sinapov, J. and Stoytchev, A. (2008). Detecting the functional similarities between tools using a hierarchical representation of outcomes. In *Proceedings of the IEEE Conference on Development and Learning (ICDL 2008)*.

Smith, R. (2006). Open Dynamics Engine (rigid body dynamics simulator). http://www.ode.org.

Sridhar, M., Cohn, A., and Hogg, D. (2008). Learning functional object-categories from a relational spatio-temporal representation. In *Proceedings of ECAI 08*.

Stilman, M., Shamburek, J.-U., Kuffner, J., and Asfour, T. (2007). Manipulation planning among movable obstacles. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA 07)*.

Stoytchev, A. (2005). Behaviour-grounded representation of tool affordances. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*.

Stoytchev, A. (2007). *Robot Tool Behaviour: A Developmental Approach to Autonomous Tool Use*. PhD thesis, College of Computing, Georgia Institute of Technology.

Stulp, F. and Beetz, M. (2008). Refining the execution of abstract actions with learned action models. *Journal of Artificial Intelligence Research*, 32:487–523.

Suc, D. and Bratko, I. (1997). Skill reconstruction as induction of LQ controllers with subgoals. In *Proceedings of IJCAI-97*, pages 914–919.

Sutton, R. and Barto, A. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

van Lawick-Goodall, J. (1970). Tool-using in primates and other vertebrates. In Lehrman, D., Hinde, R., and Shaw, E., editors, *Advances in the Study of Behaviour*, volume 3, pages 195–249. Academic Press, London.

Veeraraghavan, H. and Veloso, M. (2008). Teaching sequential tasks with repetition through demonstration (short paper). In *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS08)*.

Wang, X. (1995). Learning by observation and practice: An incremental approach for operator acquisition. In *Proceedings of the 12th International Conference on Machine Learning*, pages 549–557. Morgan Kaufmann.

Wood, A. (2005). Effective tool use in a habile agent. Master's thesis, North Carolina State University.

Yik, T. and Sammut, C. (2007). Trial-and-error learning of a biped gait constrained by qualitative reasoning. In *Proceedings of the Australasian Conference on Robotics and Automation*.

Yoshikawa, T. and Kurisu, M. (1991). Identifcation of the center of friction from pushing an object by a mobile robot. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 449–454.

Zrimec, T. (1990). *Towards autonomous learning of behaviour by a robot*. PhD thesis, Department of Computer and Information Science, University of Ljubljana.