An Architecture for Tool Use and Learning in Robots

Solly Brown and Claude Sammut
ARC Centre of Excellence for Autonomous Systems
School of Computer Science & Engineering
The University of New South Wales
Sydney, NSW 2052, Australia
Email:{sollyb,claude}@cse.unsw.edu.au

Abstract

In this paper we address the problem of a robot learning to use environmental objects as tools, in order to help it achieve its goals. Learning to use an object as a tool involves understanding which *goals* it helps an agent to achieve, the *properties* of the tool that make it useful, and how the tool must be *manipulated* in the environment in order to achieve the desired goal. A cup, for example, can be use to hold objects or liquids, should be of the appropriate size and shape (concave-up), and needs to be held the right way up. We present an architecture for a robot agent that is able to learn about objects in this way, and thereby employ appropriate objects as tools to help it achieve its goals. Our agent learns through demonstration and experiment, with the main generalisation module being an Inductive Logic Programming algorithm.

1 Introduction

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, from monkeys [de A. Moura and Lee, 2004] to bottle-nosed dolphins [Krützen *et al.*, 2005]. Although most animal tool use is instinctive and does not involve learning, some creatures such as the New Caledonian crow are adept at both tool-making and solving novel problems using tools [Kenward *et al.*, 2005]. In this paper we address the problem of tool use and learning in robots.

The motivations for studying tool use in robots are many. Firstly, tools are ubiquitous in the modern world and robots will need to understand how to use them to complete a full range of useful tasks. Robots, like humans, have limited range of effectors and can use tools to leverage their ability to solve problems in their environment.

Secondly, tool use is a specialised form of problem solving, where the emphasis is on objects, their properties, and interactions between objects, and between objects and the agent. It is unlikely a general purpose problem solver would have sufficient domain knowledge or learning bias to perform effectively in this domain.

Finally, tool use is an interesting problem to study because learning is often an essential requirement: for achieving competence, solving novel problems, understanding the properties of new objects, and learning by observing others.

Our overall research goal is to build a robotic agent which can learn to use objects in its environment as tools, in order to achieve its goals or to achieve its goals more effectively. Some examples of the kind of problems we would like our agent to be able to solve are:

- **Stick and tube problem:** A reward object is placed inside a horizontal tube lying on the ground. The agent is unable to reach into the tube directly to retrieve the reward, but instead must use a stick tool to push the reward out of the tube before it can be collected.
- **Ramp problem:** A reward object is placed on top of a platform, out of reach of the agent. The agent can get onto the platform by pushing a ramp over to the platform and driving up it. Note that this is a version of the classic monkey and bananas problem, and the same problem faced by Shakey the robot [Nilsson, 1984].
- **Tray problem:** The agent is given the task of transporting 5 small objects from one room to another, in as short a time as possible. A reward can only be achieved by transporting all of the objects simultaneously by placing them on a tray.

Of course, each of these examples is a straightforward planning problem when the agent has sufficient domain knowledge to accurately model how its actions affect the world. Shakey the robot for example, had built-in background knowledge about ramps, including a `CLIMB-RAMP`

behaviour and an abstract model of this behaviour which it used for planning [Nilsson, 1984].

In this research, however, we assume that the robot has no prior knowledge of specific properties of the tools in its environment. Thus in in the ramp problem described above, the agent starts out knowing nothing about ramps. The agent's task is to learn the necessary domain knowledge about the ramp – through observation and experimentation – so that it can then solve the problem as Shakey did.

The rest of this paper is organised as follows. In Section 2 we describe previous work that has been done on tool-use in machine learning and robotics. Section 3 describes the environment in which the robot is expected to operate. The world state, action, and dynamics representation we have adopted are then outlined in Section 4. Section 5 outlines our agent architecture, and the agent's learning methods are described in Section 6. Finally, Section 7 concludes with some discussion and an outline of our future work.

2 Related Work

Given the significant role tool use has played in research on human and animal cognition [Barber, 2003], it is perhaps surprising that it has received little attention in artifical intelligence and robotics research. Bicici [2003] provides a survey of earlier AI research related to the reasoning and functionality of objects and tools.

Perhaps the first attempt to build a robot agent specifically tailored towards learning and tool-use is given in [Wood *et al.*, 2005]. In this work an artificial neural network was used to learn appropriate postures for using reaching and grasping tools, on board the Sony Aibo robot platform.

Stoytchev [2005] has implemented an approach to learning tool-using behaviours with a robot arm. The robot investigates the effects of performing its innate primitive behaviours (including pushing, pulling, or sideways arm movements) whilst grasping different reaching tools provided by the user. The results of this exploratory learning are used to solve the task of using the tool to move an object into a desired location.

Although there is little other research that directly tackles the robot tool use learning problem, work in the area of learning models of agent actions is relevant to our approach. Benson [1996] used Inductive Logic Programming (ILP) methods [Lavrac and Dzeroski, 1994] to learn action models of primitive actions, and then combined them into useful behaviours. Other work has since focused on learning action models for planning in more complex environments, allowing for stochastic action [Pasula *et al.*, 2007] or partial observability [Amir, 2005].

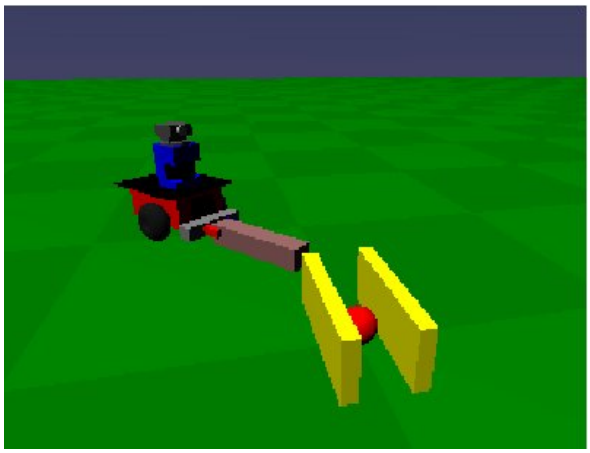

Figure 1: The Pioneer tackling the stick and tube problem in the Gazebo robot simulator.

3 Domain and Robot Platform

The agent architecture presented in this paper is intended to be sufficiently general that it can be applied to any robot. In this research our tool using robot platform is a simple Pioneer 2DX with a vertical lift and gripper attachment. Sensors include a camera, sonar array, and laser scanner[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.

We are carrying out our robot tool use experiments initially in simulation. This allows us to test ideas and carry out experiments much more rapidly than would be possible in the real world. We are using 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 Gazebo robot simulator is built on top of the Open Dynamics Engine physics engine, which provides for rigid body physics in three dimensions. Gazebo supports the Player interface and thus allows for robot control code to interact with the simulator through the same client-side interace that would be used on a real-world robot. A screen-grab of the robot tackling the stick and tube problem in the Gazebo simulator is shown in Figure 1.

[1]Our current implementation uses a noisy global GPS to get object positions and orientations; in future work this data will be obtained less directly, through the sensors.

4 World model

4.1 States

Two levels of world state information are available to our agent. At the low-level the primitive state shows the positions, velocities and orientations of the objects and agents in the world at each time step. On top of the primitive state we build an abstract state which is able to explicitly represent the complex properties and relationships which exist between objects and agents in the world.

We write the abstract state in first-order logic, a compact and expressive representation which is extensively used in planning, reasoning and learning in relational domains. Thus we write terms such as `holding(agent, stick), in(reward, tube)` to describe the state where the agent is grasping a stick and a reward object is inside a tube. The abstract state predicates are defined in terms of the value of primitive state quantities, or other previously defined abstract state predicates.

4.2 Actions

We provide the agent with a set of useful set of hand-coded behaviours which allow the agent to perform common tasks. STRIPS rules [Fikes and Nilsson, 1971] are used to model the agent's knowledge of how these actions affect the world. Given STRIPS models of its actions, the agent can use a standard planner to construct plans to achieve its goals.

STRIPS action models are composed of three lists of literals: a precondition, an add list, and a delete list. The precondition gives the requirements for the action to be applicable, while the add and delete lists specify the significant changes to the world state which will occur as a result of executing the action.

In general case, the agent begins with a set of actions $A_1, A_2, \ldots A_n$ and a corresponding set of STRIPS action models. These existing actions are insufficient to achieve the goal we present to the agent — in order to solve the problem the agent must learn to use a tool.

In the stick and tube problem, for example, the agent is provided with the `pickup(Object)` and `drop(Object)` actions, with STRIPS models as shown in Figure 2. Note that the precondition of the `pickup(Object)` action model reflects the fact that the agent is unable to pickup an object which is stuck in a tube.

pickup(Object):
PRE: empty(gripper), ¬in(Object,Tube)
ADD: holding(robot, Object)
DEL: empty(gripper)

drop(Object):
PRE: holding(Object)
ADD: empty(gripper)
DEL: holding(Object)

Figure 2: Initial STRIPS action models for the stick and tube problem. These are provided a priori to the agent, but note that they are not sufficient to solve the problem of picking up a reward which lies inside a tube.

push-from-tube(Object,Stick,Tube):
PRE: in(Object,Tube), holding(robot,Stick)
length(Stick,L_S), length(Tube,L_T), $L_S > L_T$
width(Stick,W_S), width(Tube,W_T), $W_S < W_T$
ADD: -
DEL: in(Object,Tube)

Figure 3: The desired STRIPS model for the stick tool which will allow the agent to solve problems reward-in-tube type problems. The `push-from-tube` action model is not provided a priori to the agent — it must be learnt.

5 Architecture

Our agent architecture is illustrated in Figure 4 and is comprised of the following six modules:

- Executor
- STRIPS Planner
- Observer
- Manipulation Learner
- Motion Generator
- Generaliser

The role of each of these modules is described briefly below.

The **Executor** module is straightforward: it reactively executes existing behaviours such as `pickup(stick)` at the request of the planner, and sends the resulting commands to the robot's actuators.

The **STRIPS Planner** is responsible for generating plans which can achieve the agent's goals, and to then carry out these plans by sending off the appropriate sequence of behaviours to be executed by the Executor module (in the case of existing behaviours) or the Motion Generator (for new tool behaviours).

Any standard planner can be plugged into this module for the purpose of generating plans, although some mechanism for plan monitoring and repair must be provided. We are using a simple partial order planner [Penberthy and Weld, 1992] in this manner.

The remaining four modules are directly involved in learning new tool-use behaviours and tackling the learning challenges described in the next section.

The **Observer** module attempts to identify the primary goals of the tool-use action by watching another

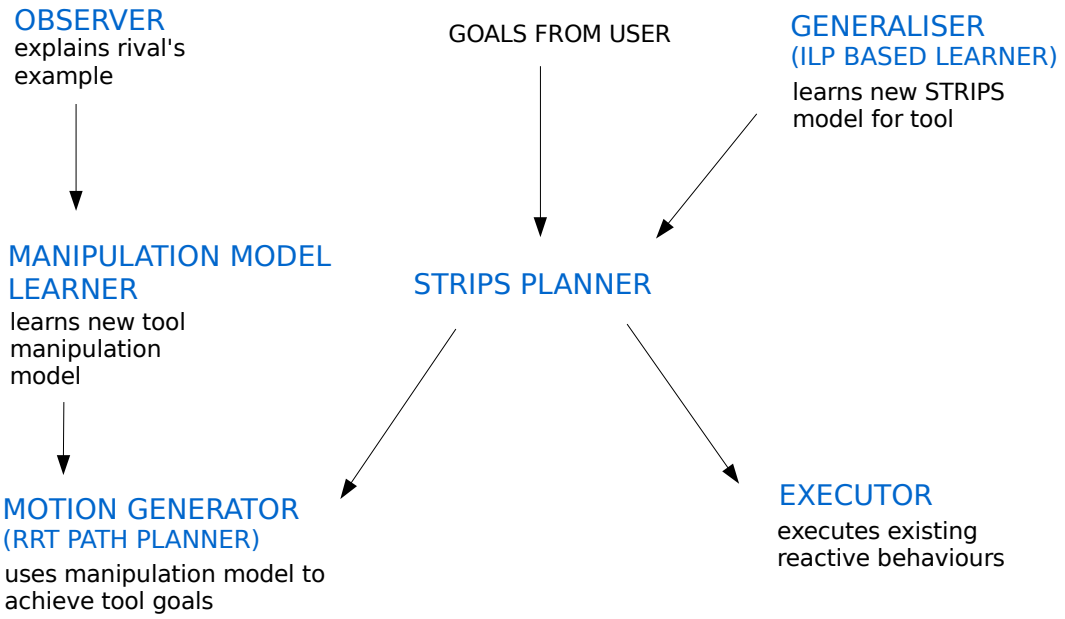

Figure 4: Agent architecture.

agent perform the task. By explaining to itself how the rival is able solve the problem, the agent can isolate the desired effects of the action.

The task of the **Manipulation Learner** module is to learn a low-level model of how the tool interacts with objects. As detailed in Section 6.2 our agent learns a set of motion primitives which characterise the way in which the tool is able to influence the object. For example, a useful set of motion primitives for the stick tool and a box shaped reward object are illustrated in Figure 7.

The **Motion Generator** module is a low-level path planning module which uses the motion primitives learnt by the Manipulation Learner module to generate more complex tool motions. It can, for example, plan a series of motions which results in the reward object being pushed from the tube by the stick tool. In Section 6.3, we describe the Rapid Random Tree (RRT) [J. Kuffner and S. LaValle, 2000] planner which forms the basis of the motion generator in our research.

Finally, the purpose of the **Generaliser** is to generalise over tools and tasks to learn the necessary properties that the tool must satisfy in order to be useful. This information is then incorporated into the STRIPS action model for the tool behaviour. As described in Section 6.4, the generalisation over first-order state and object descriptions is performed by an Inductive Logic Programming (ILP) algorithm.

6 Learning

Learning to use an object as a tool involves understanding the following:

- which *goals* the tool helps an agent to achieve
- how the tool must be *manipulated* in the environment in order to achieve the desired goal
- the *properties* of the tool that make it applicable or useful

In the stick and tube problem, for example, the desired goal is to achieve ¬`in(tube,reward)`. The stick tool must be manipulated by pushing type motions, and the required properties of the stick include being long enough to reach all the way through the tube, as well as thin enough to fit into the tube opening.

The useful effects and properties of the tool can be summarised in a STRIPS action model, such as the `push-from-tube` model shown in Figure 3. By learning and encapsulating this information in a STRIPS model the agent can make use of the tool in future planning and problem solving.

In the remainder of this section we describe how the above learning objectives are tackled, using the stick and tube problem as illustration.

6.1 Identifying tools and goals

Effective learning in the real world involves being able to watch others and emulate their success. In our learning tasks we give the agent an opportunity to observe the

actions of another agent which we will refer to as the *rival*.

The introduction of a rival agent serves to simplify the learning problem faced by our agent, by providing a demonstration which focuses attention on interesting parts of the state space. In our tool-learning tasks the rival agent is identical to the original agent, except it already knows how to use the tool to solve the problem.

In the stick and tube problem, the agent's Observer module watches the rival pickup a stick, push the reward from the tube, and then pick up and make off with the reward. The agent then seeks to explain how its rival managed to obtain the reward, and to thereby identify any novel tool-using behaviour which the agent may wish to replicate. Information about the tool used and the sub-goal achieved is then passed to the Manipulation Learner and Generaliser modules.

An outline of the algorithm we are using to construct explanations of the rival's actions is given in Figure 5. The algorithm takes as input a time-series of world states $s_1, s_2, s_3 \ldots s_n$ which describe the observations of the rival robot's activities. It tries to reconstruct the plan followed by the rival by finding a consistent mapping of its own STRIPS actions onto the time-series of states.

The algorithm begins by noting all of the actions which are applicable in the start state. For each step in the time series it then checks whether any of the currently applicable actions (those whose preconditions were true at the previous action transition) have terminated by virtue of their effects being achieved. If so, the action is added to the explanation and the set of currently applicable actions is updated.

Novel behaviours are introduced in the explanation to account for interesting state transitions which are not explained by the agent's current set of action models. In our current implementation an "interesting" state transition is defined as one in which the preconditions of an existing action are activated, but not by an existing action. In the case of the stick and tube problem, this occurs when the preconditions of the `pickup(reward)` are activated by the rival pushing the reward from the tube and causing $\neg$`in(reward, tube)`.

Using this algorithm the agent generates the following explanation of its rival's actions in the stick and tube problem:

```
pickup(stick)
?novel-behaviour(stick,reward,tube)?
drop(stick)
pickup(reward)
```

The `novel-behaviour` action is of course the stick pushing tool action `push-from-tube` the agent wishes to mimic and learn to do itself. The agent notes that this new action can be used to achieve the effect $\neg$`in(reward, tube)` and passes this information to the Generaliser model (which will later learn a STRIPS model for the action). The Observer also records the objects and tool involved in the action and passes this to the Manipulator Learning module.

generate_explanation
input: time-series of states
output: an explanation (a sequence of actions)
pre_lits ← literals which occur in any action precondition
currently_applicable ← actions applicable in start state
explanation ← {}
S_{prev} ← S_0
for each state S in time-series
  none_terminated = true
  for each action A in currently_applicable
    if Aterminates in S
      append A to explanation
      currently_applicable ← actions applicable in S
      S_{prev} ← S
      none_terminated = false
      break
  if none_terminated
    state_diff = S - S_{prev}
    unexplained_transitions = state_diff ∪ pre_lits
    if unexplained_transitions activate preconditions of
     one or more actions
      append 'novel-behaviour' to explanation
      record unexplained_transitions as effects
        of novel-behaviour
      currently_applicable ← actions applicable in S
      S_{prev} ← S
return explanation

Figure 5: Pseudo-code outlining the way in which explanations of the rival agent's actions are generated. An action A terminates in a state S if it is currently applicable and its effects are all true in S.

It should be noted that the explanation algorithm described in Figure 5 is naive in some of the assumptions it makes. In practice, we first pre-process the state time-series to remove noisy state transitions such as the jitter which occurs when a state literal flicks between being "on" and "off" (eg. when the robot begins to grip an object, the literal `holding(Object)` may appear and disappear before a firm grip is established).

In addition, it is possible that more than one action sequence might explain a given world state sequence. We therefore impose a further constraint which demands that all actions included in the explanation should support the preconditions of at least one action which occurs later in the plan. A more robust explanation algorithm is the subject of continuing work.

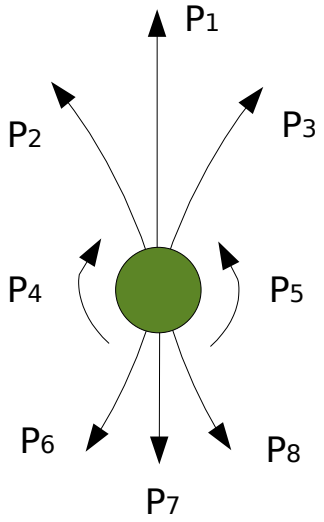

Figure 6: Robot motion primitives for our non-holonomic robot. The eight motion primitives are indicated by the arrows, comprising three forward actions, three reversing actions, and two rotate-on-the-spot actions.

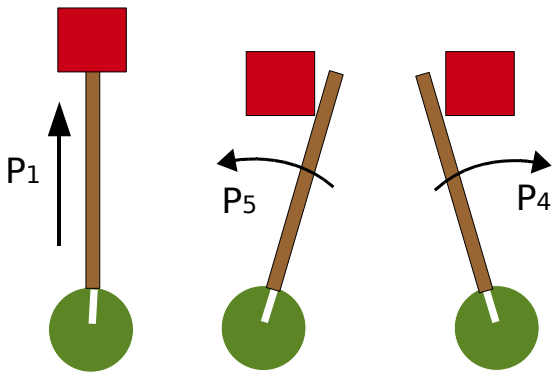

Figure 7: The three most useful motion primitives T_1, T_2, T_3 for the robot (green) and stick tool (brown) interacting with a box-shaped object (red). Shown are the initial positions of the tool and object, with the required robot primitive motions P_1, P_4 and P_5 labelled with arrows.

6.2 Learning a tool manipulation model

The agent now has a set of goals for the new action (provided by the Observer module) and the next step is to learn how to perform the necessary low-level manipulations in order to achieve these desired action effects.

We approach the problem of learning low-level manipulation of the tool and object by learning a model in the form of a set of motion primitives, and then using a low-level path planner to sequence these primitives into more complex motions.

As an example, consider the motion primitives for our non-holonomic Pioneer robot moving freely on its own, as illustrated in Figure 6. These low-level actions are provided to the robot a priori and consist of eight possible primitives in total $P_1, P_2, \ldots P_8$. They correspond to movements such as "moving forward while turning left" and "turning right on the spot", and comprise three forward motions, three reversing motions, and two rotating motions. Armed with this discrete set of motion primitives, we can use a low-level path-planner (as described in Section 6.3) to generate paths for robot motion from one point to another.

In a similar way if the agent can learn a set of motion primitives which characterise the ways the agent and tool interact with an object, we can use the same motion planner to generate useful tool-using behaviours.

The sorts of tool motion primitives the agent is trying to learn are illustrated in Figure 7, which shows qualitatively the three most useful stick tool primitive motions T_1, T_2, T_3 generated in the case of the stick tool acting on a box-shaped object.

These tool primitives are learnt in a relatively straightforward manner by placing the stick tool in contact with each of the surfaces of the object and observing the result of executing each of the agent's own primitve motions $P_1, P_2, \ldots P_8$. For example, the move-forward robot primitive P_1 results in the straight-forward push primitive T_1 (shown in the figure) when the stick end is placed directly in front of the object. In contrast, the same robot motion P_1 produces a rather less interesting tool manipulation primitive if the object initially lies alongside the stick — in this case, the object barely moves.

Internally, the agent represents each tool primitive as the average vector displacement ΔToolPose, ΔObjectPose which results when a given robot motion primitive P_i is executed at a specified tool-object contact point and orientation.

Because of the variety of different ways in which a tool and object can interact a large number of tool primitives can be generated. Many of these are uninteresting and we therefore cull the set of primitives by removing those which have a negligible effect on the resulting position of the object. A maximum limit of eight characteristic motion primitives is then imposed by removing those which have a similar effect to others already included in the set.

$\mathbf{e}_1$: **neg**, < length(stick1, 0.3), width(stick1, 0.15), length(tube1, 0.6), width(tube1, 0.2), colour(stick1, blue),. . . >
$\mathbf{e}_2$: **pos**, < length(stick2, 0.8), width(stick2, 0.12), length(tube1, 0.6), width(tube1, 0.2), colour(stick2, red),. . . >
$\mathbf{e}_3$: **pos**, < length(stick3, 0.6), width(stick3, 0.13), length(tube2, 0.5), width(tube2, 0.18), colour(stick3, red),. . . >
$\mathbf{e}_4$: **neg**, < length(stick3, 0.6), width(stick3, 0.13), length(tube3, 0.5), width(tube3, 0.12), colour(stick3, red),. . . >
Correct concept:
length(Stick, L_S), length(Tube, L_T), width(Stick, W_S), width(Tube, W_T), $\mathsf{L}_S > \mathsf{L}_T$, $\mathsf{W}_S < \mathsf{W}_T$

Figure 8: Some examples for the "useful-tool" concept learning problem, along with target concept. The label (pos/neg) is followed by the (truncated) state vector describing the example.

6.3 Using the tool

Once the agent has built up a small repertoire of useful motion primitives, it can use a low-level path planner to use the tool to try and achieve the goals supplied by the Observer module (eg. achieving $\neg$in(reward, tube)).

The motion planner we are using in this research is a Rapid Random Tree (RRT) path planner [J. Kuffner and S. LaValle, 2000]. An RRT planner is ideally suited to this problem because it can be fed an arbitrary set of motion primitives with which to plan a path. Errors in the motion primitive model mean that tool-use paths generally cannot be followed in an open loop manner, but the speed of the RRT-based planner allows fast re-planning.

6.4 Learning tool properties and a STRIPS model

Having learnt to use the tool to solve the problem at hand, the agent must now try and learn the properties that make this type of tool useful. In the stick and tube problem, for instance, a stick is only useful for extracting the reward if it is long enough to reach all the way through the tube, and thin enough to fit into the tube in the first place.

The necessary generalisation across tools and objects is performed by the Generaliser module, as described in this section. The required properties of the tool can then be incorporated into a STRIPS action model, as shown in Figure 3, and the tool can be used in future problem solving.

In order to be able to generalise effectively we present the agent with a series of different instances of the same type of problem. For the stick and tube problem this involves using different sized tubes and a selection of different potential stick tools.

Identifying which properties make a stick a useful tool for this task is a concept learning problem. A stick with which the agent is unable to solve the task is a negative example of a good tool for that task, and conversely a stick which succeeds in extracting the reward is a positive example.

Some examples defined in this way for our stick tool problem are shown in Figure 8, along with the correct concept to be learnt — which includes the conditions $L_S > L_T, W_S < W_T$ that express the fact that the stick must be longer and thinner than the tube in order to be useful.

The concept learning problem is relational in nature because a correct description of the concept involves relationships between objects (eg. the stick must be longer than the tube) rather than simple attributes. Relational concept learning has been extensively studied in the field of Inductive Logic Programming (ILP) [Lavrac and Dzeroski, 1994]. Our learner uses the ILP algorithm Aleph [Srinivasan, 2001], a derivative of Progol [Muggleton, 1995], to learn the concept which describes a good tool for the task. The reader is referred to Muggleton's paper for a detailed description of the algorithm.

7 Conclusions and Future Work

In this paper we have presented an architecture for tool use and learning in robots. It allows an agent to learn about objects in its environment and to identify the properties and manipulation necessary in order to employ them as tools. Our agent learns primarily through direct interaction with the environment, with demonstration by a teacher used to provide a starting point for learning.

The novel contributions of this research are, firstly, a general architecture and approach to learning to exploit objects as tools in a three dimensional world, in an online incremental manner. Previous work on tool use in robots has been considerably less general, and has focused purely on learning tool manipulation. Here we incorporate tool-use into a problem solving context, and describe ways of generalising in order to learn the properties which make a tool useful.

Another contribution of this research is to show how robot agents can autonomously acquire new behaviours and incorporate them into their high-level problem-solving — without requiring the user to explicitly define the new behaviours to be learnt.

In ongoing work we are evaluating the architecture described in this paper by carrying out tool use learning trials in the Gazebo robot simulator. Our aim is show that the approach presented here is capable of solving a wide range of tool-using problems, including the ramp and tray problems described in the introduction.

Acknowledgements

This work has been supported by the Centre for Autonomous Systems and National ICT Australia.

References

[Amir, 2005] E. Amir. Learning partially observable deterministic action models. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 1433–1439, Edinburgh, Scotland, UK, August 2005.

[Barber, 2003] Chris Barber. *Cognition and tool use: Forms of engagement in human and animal use of tools*. Taylor & Francis, July 2003.

[Benson, 1996] Scott Benson. *Learning action models for reactive autonomous agents*. PhD thesis, Department of Computer Science, Stanford University, 1996.

[Bicici and St Amant, 2003] E. Bicici and R. St Amant. Reasoning about the functionality of tools and physical artifacts. Technical Report 22, NC State University, 2003.

[de A. Moura and Lee, 2004] A.C. de A. Moura and P.C. Lee. Capuchin stone tool use in caatinga dry forest. *Science*, 306(5703):1909, December 2004.

[Fikes and Nilsson, 1971] R.E. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[Gerkey *et al.*, 2003] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. 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, June 2003.

[J. Kuffner and S. LaValle, 2000] J. Kuffner and S. LaValle. 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*, April 2000.

[Kenward *et al.*, 2005] B. Kenward, A.A.S. Weir, C. Rutz, and A. Kacelnik. Tool manufacture by naive juvenile crows. *Nature*, 433(121), 2005.

[Koenig and Howard, 2004] N. Koenig and A. Howard. 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, September 2004.

[Krützen *et al.*, 2005] Michael Krützen, Janet Mann, Michael R. Heithaus, Richard C. Connor, Lars Bejder, and William B. Sherwin. Cultural transmission of tool use in bottlenose dolphins. *Proceedings of the National Academy of Sciences*, 102(25):8939–8943, 21 June 2005.

[Lavrac and Dzeroski, 1994] N. Lavrac and S. Dzeroski. *Inductive logic programming: Techniques and applications*. Ellis Horwood, 1994.

[Muggleton, 1995] S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

[Nilsson, 1984] Nils J. Nilsson. Shakey the Robot. Technical note 323, SRI International, Menlo Park, CA, April 1984.

[Pasula *et al.*, 2007] Hanna M. Pasula, Luke S. Zettlemoyer, and Leslie Pack Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352, 2007.

[Penberthy and Weld, 1992] J. Scott Penberthy and Daniel Weld. UCPOP: A sound, complete, partial-order planner for ADL. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Proceedings of the third international conference on knowledge representation and reasoning (KR-92)*, pages 103–114, Cambridge, MA, October 1992. Morgan Kaufmann.

[Srinivasan, 2001] A. Srinivasan. The Aleph Manual. http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph.html, 2001.

[Stoytchev, 2005] A. Stoytchev. Behaviour-grounded representation of tool affordances. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, April 2005.

[Wood *et al.*, 2005] A.B. Wood, T.E. Horton, and R. St Amant. Effective tool use in a habile agent. In *Systems and Information Engineering Design Symposium*, pages 75–81. IEEE, April 2005.