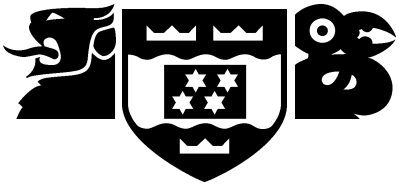
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

An Agent That Acts Based On Past Experience

Alan Longhurst

Supervisor: Peter Andreae

October 2004

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

This report introduces the RPLA system (Reactive Planning Learning Agent), a simulation-based agent that learns rules from experience in the world that are used reactively to control the agent's behaviour. The rules are adapted to fit with the current goal, so rules learnt to satisfy one goal can be applied to new goals. A novel matching algorithm is used achieve this effect and to allow for differences between the rule and the state of the world so that the world can change after the agent has learnt the rules and the rules will be able to be used. A set of tests show that the system is able to perform a variety of tasks using the rules it has created.

Acknowledgements

First and foremost I would like to thank my supervisor, Peter Andreae, for his support throughout this project and for not stressing *too* much one week before the deadline.

I would like to thank my colleague, Maciej Wojnar, for the discussions on various aspects of this work.

I would also like to thank the denizens of Memphis for the interesting debates on pointless subjects, *even though they abandoned me when the report was due*.

Contents

Figures

Chapter 1

Introduction

1.1 The goals of the RPLA system

One of the goals of Artificial Intelligence is to build autonomous intelligent agents that can act and interact in the real world. There are many different aspects of intelligence that are required for an autonomous intelligent agent; the ability to learn about actions from experience is central.

The standard way of representing knowledge about actions is some form of rules. There are many systems that use rules for planning and choosing actions. There has been much less work on systems that learn such rules from experience

The goal of this project is to develop an agent that can learn good (but not necessarily perfect) rules from experience. After training, the rules will be used by the system to reactively select the actions to achieve new tasks. This requires that the rules constructed by the system are able to work in the future, where both the world and the goal may be different. Because the teacher's time is valuable, the agent must learn quickly – only a few examples will be provided for the system to construct rules from. So that more complicated rules can be learnt, and basic ones can be improved, the system must be able to adapt the rules as it receives more experience.

This report introduces the RPLA system (Reactive Planning Learning Agent), a simulation-based agent that learns rules from experience in the world that are used reactively to control the agent's behaviour.

There are various activities related to learning rules for actions to which RPLA could contribute, but are not the focus of this project. RPLA learns rules, it does not address the following:

- Exploration: the teacher shows the actions that are required; the agent does not attempt to discover how to solve something, but assumes it has been provided with the examples of what is required.
- Planning ahead: the agent selects actions as they are needed, not ahead of time. The system does not look ahead at the actions it might choose nor does it look it the past at actions it has chosen.
- Learning about effects: The system does not consider what the actions do, only what action should be performed.

- Learning optimal behaviour: agent chooses actions based solely on what fits best to the current situation, not on the expected benefit of that action. The measure of the system is its ability to learn the rules and adapt. The 'quality' of the plan produced is not considered.

1.2 Outline of Report

The report is structured as follows: The next chapter gives background into the concepts used in this project, and Chapter 3 presents the simulated world that is the home of the agent. Chapter 4 describes how the agent applies the rules to select actions to perform, Chapter 5 details the process of creating a rule from an example followed by the methods used to improve those rules in Chapter 6. Chapter 7 presents the results from tests of the functionality of the system and discusses some limitations identified. The paper concludes with a discussion of future work to combine the RPLA system with other work currently in progress.

Chapter 2

Background

2.1 Traditional Planning

The task of planning is a well established domain in artificial intelligence research and much work has been done in this area. There are many different approaches used today, however they almost all use the same general concept: The planner is given a goal and the description of the current state and creates the plan by starting at the goal and working backwards – each step in the plan is decided by assessing an unachieved state of the world and choosing the action that will result in that state. Planning in this manner has its advantages, for example it will be clear before the agent starts performing actions whether or not that agent can actually achieve the goal – if the agent can create a valid plan for the task than the agent can achieve the goal. It also means that once the agent begins performing actions there is very little processing required to perform the next action.

Unfortunately, this strategy is insufficient for many applications where there are other considerations that complicate the process. The accuracy of the plan generated is dependent on the accuracy of the rules the planner uses to create the plan. To create an entire plan from start to finish, the agent must have all the information it needs at the beginning. The rules the planner uses must also be reliable and complete. If the agent is missing some rules then the planner simply fails. Even worse than failing before starting the plan, the plan may break down if the rules used are not accurate and so the outcome of some action when performed is not what the agent expected. If the agent is following a path and at some point the agent finds it has veered off that path then it will not know what to do next and some action must be taken to remedy the situation. Planning backwards also means that the first action the agent must perform is decided after later actions – the first actions chosen are usually the last to be performed. Considerable planning is performed to get to the 'start' of the plan, so the entire plan must be created and verified before the agent can actually begin executing the plan. The planning process can take a considerable amount of time to produce a consistent plan and in some implementations the time required increases exponentially as the complexity of the task increases. Finally, planning backwards is not the technique generally used by people to solve problems [1].

Consider the task of making a cup of coffee. Using the traditional planning approach the agent will start with the final product – the hot cup of coffee – and decide the actions it needs to get there. For example, the last action performed could be "stir the drink with a spoon",

[1] Reverse-engineering can be useful to discover how someone else solved the problem, but this would not be how the problem was solved in the first place.

the action before that might be "add milk/sugar". Each action in the sequence is decided by considering actions later on in the sequence.

2.2 Reactive Planning

A different approach to planning has been proposed [4] that avoids many of the problems described above and better reflects the way in which we solve routine tasks. Instead of working backwards from the goal to the current state, *reactive planners* work forwards towards the goal. Each action (or action set) the agent performs is chosen as it is needed. The agent chooses actions by assessing the current state of the world and using rules to select actions that it believes are best given the current situation.

Consider again the coffee example. Rather than starting at the end, a reactive planner begins by decided on the first thing to do – find the cup/coffee. Once this is done, it decides on the next course of action. This is a more natural process and I conjecture is the method most people would use. Perhaps more importantly, reactive planners correspond to the way we are taught tasks. Consider a teacher and a student; the teacher wants to show the student how to perform a task. The teacher will give step-wise instructions to the student, ensuring that the student has successfully completed the last step before giving the next instruction or, just as significantly, taking some other action, such as rephrasing the instruction, if the student could not perform the last requested action successfully. When the student repeats that task some time in the future, the student will follow the same procedure. The student will assess what has been achieved and decide what needs to be done next, rather than what is left to be done. A reactive planner operates in the same way, thus it is a natural approach to use for an agent that is intended to act like a person, since people rarely perform the complicated planning used by traditional planners.

A reactive planner will take information about the current situation and use its rules to select the next action to perform. Each time an action is required, the planner will have up-to-date information about the world. This approach provides the following benefits over traditional planning:

- When using a reactive planner, the agent selects the first action to perform by evaluating the state of the world against the rules it possesses. The next actions do not have to be decided straight away, but can (and should) be dependent on the outcome of the first action. This lowers the response time of the agent (the time from when a job is requested to when some response is first received) since the agent only needs to decide upon the first action to perform before it acts rather than creating and verifying the entire plan sequence.
- By making the selection of actions independent, the hard problem of achieving some complicated goal is separated into multiple, easier tasks of deciding one action to perform. Mechanisms may need to be added, however, to avoid plans looping back onto themselves just as they are needed in traditional planning.
- Reactive planning is much more resilient to incomplete rules. Planning can occur even if the rule set the agent possesses may be incomplete or unreliable. The agent does not need to have a fully determined plan before performing actions, so the agent can perform some actions and then wait for better information before continuing.

2.3 Reactive Rules

Any planning system requires rules of some description to make assessments about the actions to perform. The difference between reactive planners and traditional planners is how the rules are applied. Reactive planners do not need information about the effect of the actions, as the outcome of one action is not used to plan the next actions; the information required by the planner will be provided to the planner by examining the world. Reactive rules must refer to the current situation, and specify the action to perform, but do not need to say anything about the state of the world after the action has been performed. The basic reactive rule will therefore look like:

> "If the world matches X, then perform action Y."

Rules in this basic form can be used reactively to select actions to perform. However, the agent using these rules will only be able to solve a single task. Because the rules are independent of the task at hand, the rules will always specify the same action to perform given the same state of the world. This limits the adaptability of the agent and the rules. To get the agent to perform new tasks, the agent will need a new set of rules. To address this issue, the approach proposed in this paper incorporates the goal into the rules used by the agent. By matching the goal in the rule to the current goal, the agent can not only select the rules appropriate to its current goal, but it may also be able to modify the action suggested by a rule to reflect the task at hand. With the goal included, the form of the rules used by the agent becomes:

> "If the world matches X and the goal matches Y, then perform action Z."

2.4 Reinforcement Learning

Selecting actions based on the current state of the world has some parallels with Reinforcement Learning (RL). The general principle of RL is that the agent learns the value of being in each state of the world from feedback received during experiments in the world [16]. Once these values of the world are known the agent has learnt a policy for action selection that will enable it to get to the highest value state. At each state, the agent decides which action to perform by examining the neighbouring states, and chooses the action that will lead it to the highest value. Thus a plan created by a reinforcement-learning system is similar to reactive planner in that the action choice is determined by the current state of the world.

The major drawback of RL that makes it unsuitable for the purposes here is that the policy learnt by the agent is directly dependent on the state of the world used to learn it and will not be able to be used when the world changes. RL learns a model of the world that shows the value of being in each state. Change the world slightly and these values could become completely out of sync with the true value of those states. If the world is changed the agent must undergo the learning procedure again to learn the correct values for this new world. This problem is even more serious if the goal of the agent is changed since the values in the model learnt by the agent direct that agent to achieve one goal, they will not be of any help when the goal state is altered

2.5 Learning the rules

2.5.1 Human-Constructed

The planner uses rules to select the actions, but how should the system get these rules? The simplest approach is for a programmer to manually encode these rules into the agent. This requires the programmer to have intricate knowledge about the world, the agent, and the tasks the agent will be given. Creating rules by hand that will always work in the right situations and never in the wrong ones is a difficult process. The rules must make the right selection of action based on the information provided. As the world changes, rules that were created to fit with one situation may not work perfectly with new situations. This gets even more difficult as the number of rules increases. The more rules the agent possesses, the more chance there is for the rules to conflict with each other. A general rule that applies in all situations is adequate if the agent should always perform the same action. If the agent must perform different actions depending on the situation, the rules need to be constructed so that they only work for their section of the state space. Adding more rules could require altering the previous rules so that they fit within this constraint. A set of rules written by a programmer that operate perfectly together could fail when new rules are added to the set.

2.5.2 System-Constructed

A different approach is to have the agent construct the rules itself. Based on the agent's experience in the world, it creates the rules that it will use to select actions in the future. To learn how to accomplish a task, the agent is shown the actions to perform by an expert. The agent can then create rules based on these examples that will guide the agent to perform the same actions the expert performed, enabling the agent to successfully accomplish the task under self-direction.

Learning the rules automatically removes the need for the programmer to develop the rules. The programmer does not need to consider the agent's domain or the tasks it will be given as the rules the agent constructs will deal with this implicitly.

Teaching the agent new tasks uses the same procedure, so more knowledge can be added to the agent at any time. This allows the agent to cope with tasks that it was not originally taught. Simply showing the agent the new task is enough for it to learn how to perform it.

Finally learning from experience closely mimics the way people learn about the world. People are not provided with a set of rules that they apply; people learn rules by interpreting their experience.

2.5.3 Learning with a Teacher

Learning about the world without instruction is too hard, even for people. Without any direction, there is little hope that the system will learn anything useful about solving tasks. The behaviour will resort to random action selection, which may eventually happen upon the goal, but creating reactive rules from this would be hopeless. The project is focussed on the problem of learning from experience, not on exploration techniques, therefore a teacher is used to demonstrate the actions that are required to the agent.

2.5.4 Issues with Learning from Experience

Learning from experience is an attractive and well-used technique, however it introduces some complications that must be considered. A simple example of a real world task illustrates these issues:

Jimmy is being taught how to wash dishes. His father takes a plate, puts it in the water, washes both sides of the plate, and put then puts the dish rack to dry. Jimmy will copy this procedure for another plate. Jimmy could then the same procedure to wash any plate, and should be able to adapt the technique to work with bowls, cups, cutlery, and pots. Of course, Jimmy will probably try to wash an electric frying pan in the same manner, at which point his father will have to intervene [2]. This illustrates one of the problems with learning from experience and allowing free adaptation - some situations are exceptions to the rule, and supervision is required to learn when the rule can not be used as expected.

Because a teacher is needed to learn, it is desirable that learning can occur with as few examples as possible. The teacher's time is assumed to be highly valuable, so it should not be wasted showing repetitive examples of the same concepts. Learning from a minimal number of examples means that the rules may not be correct. The agent must be able to identify when the rule construction has made an error and remedy the situation. More complicated tasks may require more examples to create accurate rules. The agent must be able to learn from any minimal information but must also be able to take advantage of additional information as it is provided. In practice this means the agent will improve its rules as necessary when it encounters additional examples. Obviously there is a trade off between the number of examples used to create the rules and the accuracy of the rules.

2.6 Related Work

2.6.1 Reactive Planners

The concept of a planner using the current state of the world was first proposed by Brooks in [4]. Rather than creating a model of the world and generating plans from that model, Brooks' approach was to use the world itself as the model and let the behaviour of the system be governed by that. Systems that used this concept became known as *reactive systems*.

This system spawned a number of new robot controllers that could navigate around a world effectively, but the approach did not deal with solving complex tasks. Firby [8] introduced an architecture known as *Reactive Action Packages* (RAP) that used the state of the world to decide the actions to perform to achieve. The RAP architecture indexes a set of actions by the goal, then the action to perform is chosen from this set based on the details of the world. This form of architecture is very similar to that which will be used by the RPLA system. One of the considerations taken into account by Firby was time constraints. A number of details about the time required for actions are included in the action description, and selection of an action considers these details along with the state of the world.

The RAP architecture or concepts derived from it has been used by many systems to create agents that can produce action choices in a restricted time frame. This is often combined with another sort of planner that operates in unison with the reactive system. [14] is a system that has been designed in this way. Given only a small amount of time, the system

[2] Electric frying pans generally should not be submerged

will act reactively. However, when more time is available, the system will perform more complex planning of the actions, using the reactive rules to guide the planner's search.

Some of these systems deal with similar problems as RPLA of using reactive rules in a complex world, but to the author's knowledge, none of the current systems learn these rules from experience, but rather are given the by the programmer. The RPLA system will demonstrate that these rules do not need to be provided to the agent but can be learned as the system is used.

2.6.2 Representation

The representation RPLA uses for the objects in the world is based on that developed by David Andreae as part of his PhD thesis [2]. He showed that a structural representation of objects without a full first order representation of the relationships was adequate for a system to perform matching of objects without domain-specific knowledge. Using this representation can also improve the efficiency of matching and generalisation algorithms. RPLA borrows from this work, however the system uses a much simplified approach. Although this simplified version may prove to be inadequate for the learning that is desired, additional complexity should not be introduced into the system if it is unneeded.

David Andreae's work built on earlier work by Patrick Winston, where a full first order structural representation of objects was used. Winston's vision system [18] described objects in the world by their properties and relationships to other objects, very much like the representation used by RPLA. Unlike RPLA, the descriptions of objects used by Winston's system refer to real objects in the world, however the representations are sufficiently similar that many of the principles used by Winston are directly applicable to RPLA.

As well as the representational similarities, many of the learning techniques and heuristics employed in the RPLA system also have a connection to those used in Winston's system. Winston's system learns about visual concepts with the aid of a teacher and addresses a number of the issues relating to learning from experience. The system is presented with a configuration of objects in the world, and classifies the configuration as one of a collection of concepts. The teacher tells the system whether or not it made the correct decision and the system modifies its definition of the concepts based on this feedback. This presents an excellent example of a system that is required to learn using the same process as RPLA.

Other forms of representation have been used by systems in a similar domain. Finney et al [7] used a *Deictic* representation ([1]) for a reinforcement learning algorithm. It was shown that this approach was detrimental for the learning task used, rather than improving it. The system using a deictic representation performed worse than the system using a full-propositional representation. The failure of this experiment was probably caused by both the learning technique used – direct reinforcement learning – and the particular design of the deictic representation. This report will show that using the right representation, deictic approaches can be used for learning tasks.

2.6.3 Learning from Experience

The work by Winston has already been discussed in Section 2.6.2. Winston's system uses a number of techniques for learning that are included in RPLA.

Various other systems have been developed to learn from the environment. The most directly related work evaluated was the LIVE system by Wei-Min Shen [15]. The LIVE system

learns rules about its world from experience that it will use to achieve goals. However the rules are not reactive rules, but a more standard STRIPS-style [6] that are used by a backward-chaining planner. The system will first attempt to generate a plan to solve a task, and if the planning fails or mistakes are discovered in the rules, the system will explore the world more to develop more knowledge about how it works. The system does not develop reactive rules in the same sense.

Chapter 3

The Agent's World – Craneworld

3.1 Overview

This chapter details Craneworld, the simulated world used by RPLA. Craneworld was created by David Gilligan, a previous Victoria University research assistant, to provide a rich simulated environment for the development of learning agents.

The system enables agents to view the current state of the world, perform actions on the world, and see how the world changes. Figure 3.1 shows the basic procedure of the simulator. Craneworld has been designed to allow easy extendibility. The idea is that the world can be made arbitrarily complex by defining more objects, actions, relationships and properties. There is no limited placed on the numbers of any of these things.

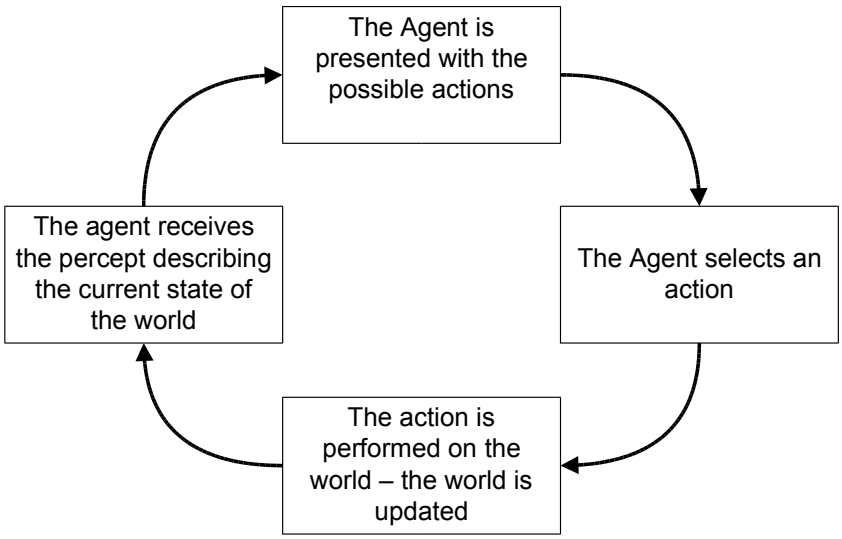


Figure 3.1: The Simulation Cycle

3.2 Objects

Everything in the simulated world is an Object. Objects are identified by their name, their type, a list of properties and a list of relationships. There are numerous objects already defined in the world, and more can easily be added. Figure 3.2 shows a simple setup of the world with a large group of objects. Objects are non-divisible, so there is "a piece of water", that can be manipulated just like any other object.

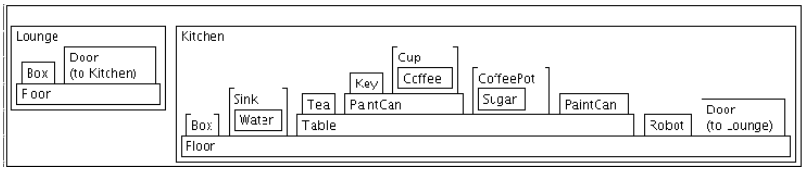


Figure 3.2: Sample World

3.2.1 Type

Every object in the system is a specific type. The type of the object is "what the object is". For example, all cups in the simulator (Cup1, Cup2, etc) are of the type *Cup*. Identifying the type of an object is important because different objects can be used for different things depending on their type. For example, a *PaintCan* can be used to change the colour of an object, whereas a *Key* can not.

3.2.2 Attributes

Objects in Craneworld are described by their individual properties and by their relationships with other objects. Collectively these two components of an object are referred to as the object's attributes. All attributes are free to be changed during the course of the simulation, provided the necessary action is performed on the object.

3.2.3 Properties

Properties give details about an object. These describe the state of the object in terms of certain measures. Common properties used are:

- Colour
- Locked (yes/no)
- Open (yes/no)
- NeedsKey (yes/no)

3.2.4 Relationships

The connections between objects are represented by relationships. The relationships implemented in the system are:

- Supports
- Contains
- On
- In

The four relationships are actually two concepts; the second two relationships are simply the inverse of the first two relationships. Both side of the relationship are recorded in the objects in Craneworld. If one object (Object A) has the relationship "supports: Object B", then Object B will have the relationship "on: Object A". Objects can *support* or *contain* any number of other objects; the object 'grows' to accommodate. On the other side, an object can only be *On* and *In* one object at a time. The relationships are not transitively defined, even though they behave as if they were. If Object A is in Object B, and Object B is in Object C, then even though Object A is "in" Object C, this relationship will not be represented in the system.

Figure 3.3 shows a simple setup of the world with only the robot and four other objects. The details of the objects are shown in Figure 3.4.

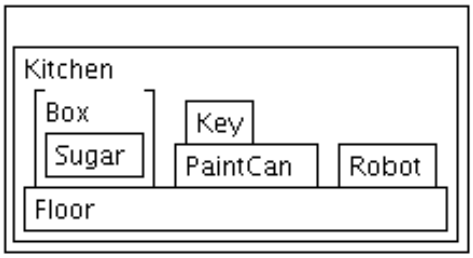


Figure 3.3: Small World – A basic configuration with four objects and the Robot

3.2.5 The Robot

A special object in the world is the robot. The robot must obey all of the rules adhered to by all objects in the world, and so acts like a normal object. However, it additionally provides the means for interaction with the world. The robot is essentially the object that performs the actions on the world, so all the actions describe something the robot should do. For example, the action of "Pick up *Sugar*" is implicitly "Put *Sugar* in *Robot*". Just like other objects, the robot can hold and support an unlimited number of objects. The robot has no explicit hands, but holds objects by *contain*ing them.

3.2.6 The Agent – RPLA

The RPLA system is an agent in Craneworld. The agent dictates the actions that the robot will perform, and views the world from the perspective of the robot. The robot represents

Name: sugar1
Type: Sugar
Property: Colour white
Relationship: In box1

Name: box1
Type: Box
Property: Open Yes
Relationship: On floor1
Relationship: Contains sugar1

Name: robbie1
Type: Robot
Relationship: On floor1

Name: key1
Type: Key
Relationship: On paintcan1
Property: Locks Any

Name: paintcan1
Type: PaintCan
Property: Colour Green
Relationship: On floor1
Relationship: Supports key1

Figure 3.4: The details of the objects in Small World

the agent's interface with the world. Symbolically, the robot is the body and the agent is the brains. In this report, the robot and the agent will be treated as the same entity, because for the purposes of the RPLA system there is no reason to make the distinction.

3.3 Perceptions

Perceptions are a textual description of the objects given to the agent. A perception is intended to describe the state of the world as the robot sees it – only the details that can be visibly identified are given in the perception. The important implication of this is that the agent is not told the identities of the objects – the names (cup1) – as this is not something that can be visually determined [1]. Figure 3.5 shows the perception the agent would receive if the world was in the state shown in Figure 3.3. Although not completely correct, for the rest of this report, 'state of the world' refers to the perception that would be created for that state.

(type: [Robot]) (On: [Floor])
(type: [Sugar])(Colour: [white])(In: [Box])
(type: [PaintCan])(Open: [No])(Colour: [Green])(On: [Floor])
  (Supports: [Key])
(type: [Key])(Colour: [Iron])(On: [PaintCan])(Clean: [yes])
(type: [Box])(Open: [Yes])(Colour: [Orange])(Clean: [yes])
  (Contains: [Sugar]) (On: [Floor])

Figure 3.5: The perception of the objects in Small World

[1] unless someone has written the names on the objects

3.4 Actions

Actions are the operations the robot can perform on the world and are the way the robot changes the state of the world. The following actions are implemented in Craneworld:

- Close
- Lock
- MoveThrough
- Open
- Operate
- Paint
- Pickup
- PutDown
- PutIn
- Unlock

Each type of action has a specific number of arguments, either one or two, that identify the objects the action will be performed on. For example, the "PutIn" action has two arguments – the object to be moved, and the object that will be the container.

3.5 Goals

Goals are an important component of the simulation for RPLA. Goals are represented in the simulator by descriptions of imaginary objects, called *goal objects*. These descriptions are consistent with the descriptions for real objects in the world – goal objects are described by a name, a type, properties and relationships. From the agent's viewpoint, the goal is seen as a perception of some objects, just like perceptions of real objects. When goals are shown in this report, the agent's view (the perception representation) is given. The description of goal objects can include a special token, *dontcare*, which indicates that the value of an attribute is unimportant. For all goals given to RPLA, the name of the goal objects is set to *dontcare*, meaning that the identity of the object is unimportant. An example goal is shown in Figure 3.6.

3.5.1 Goal-satisfying Objects

The descriptions of the objects in the goal are the requirements that must be met for the goal to be satisfied. The goal is satisfied when there exist objects in the world that subsume the descriptions of the objects in the goal. That is, for each of the objects in the goal, there is an object in the current state of the world that has all of the same attributes as that goal object. When the *dontcare* token is used, the attribute is deemed to match with all values. When current state objects are paired up with goal objects in this manner, the current state objects are labelled *goal-satisfying objects*. These objects are the objects that show the goal is satisfied.

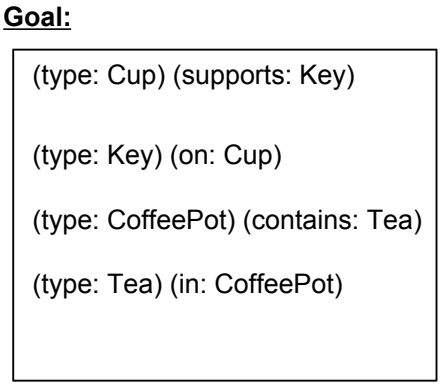


Figure 3.6: An example Goal

3.6 Time

The agent perceives the world in discontinuous intervals. There is the state of the world before the action is performed – the *before-state* – and the state of the world after the action has been performed – the *after-state* [2]. The actions have no duration, and the agent does not 'see' the action being performed; they simply happen. Conceptually this is as if the agent looks at the world (the before-state), shuts its eyes and performs the action, then opens its eyes and sees the effect. The agent discovers the changes to the world by comparing the before-state with the after-state.

3.6.1 A Step of the Simulation

A single step of the simulation is the full process required for one action to be performed. This starts with the agent taking a perception of the world – the before-state. Using this information the agent chooses an action and this is performed on the world. The agent then takes another perception of the world to see the effect of the action – the after-state – ending the step.

Steps overlap because the before-state of one step is the same as the after-state of the previous step; similarly, the after-state of the current step becomes the before-state of the next step.

When each step is performed, the agent is aware of the current goal. The goal will remain the same for a number of steps – the steps required to accomplish that goal. The teacher will set a new goal to achieve in between steps, presumably, although not strictly necessarily, after the old goal has been achieved.

A step thus is identified by four components:

- The before-state of the world
- The goal that is being achieved

[2] Remember, the before-state and after-state are really referring to the perception the agent receives of the state of the world

- The action performed, as chosen by the instructor
- The after-state of the world

Chapter 4

The Rules of the System

The RPLA system learns reactive rules from its experience. This chapter describes the form of the rules that RPLA learns, and describes how the rules would be used to drive the agent's behaviour.

4.1 The form of the rules

As introduced in Chapter 1, the rules used by the agent take the form of "If the world matches X and the goal matches Y, then perform action Z." The rules have three separate parts: the current state and the goal, which together form the condition of the rule, and the action that is the outcome of the rule.

The state of the world in a rule will be referred to as the 'rule state', primarily to differentiate it from an actual state of the world in the simulation. A rule state is very similar to a state of the world in Craneworld, with three important additions to facilitate the matching required by the system. First, the attribute values of the objects in the rule state can be a special token, *any*, that matches with any object in the world. The *any* token is used to represent the fact that some object should be present, but the details of an attribute of the object are unimportant. The second addition is tags or markers placed on objects in the rule state to connect them to objects in the goal. The final modification is 'must be' and 'must not be' tags that can be placed on objects, attributes and attribute values. The significance of each of these changes will be explained in the following sections.

The goal in a rule has the form described in Section 3.5. A goal recorded in the rule is identical to a goal of the agent, but will be referred to as a rule goal, to distinguish the goal in a rule from the agent's current goal

As with the rule goal, the action in a rule is identical to an action in Craneworld, and includes the action type and the objects that are arguments of the action. The arguments are specified by tags pointing to the relevant objects in the rule state.

An example rule is shown in Figure 4.1.

4.2 Choosing a Rule to Apply

The agent uses the rules to select the action to perform in each iteration of the world. The agent will have a collection of rules, so each time the agent needs to perform an action, it

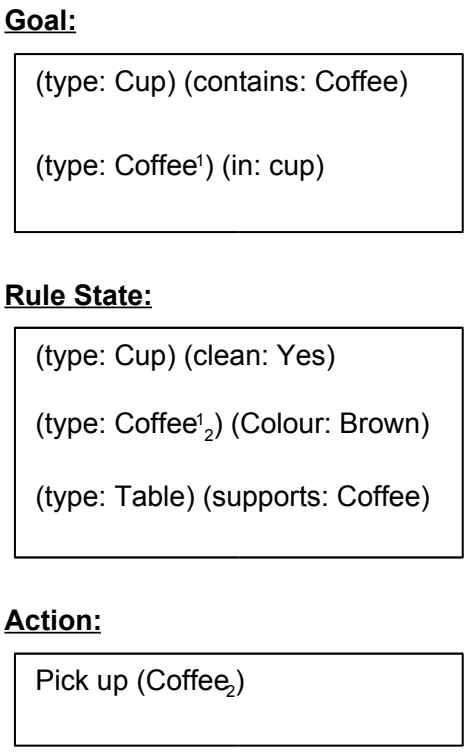


Figure 4.1: Example Rule

assesses each of its rules against the current situation to find the best rule to apply. The action specified by that rule is the action the agent will attempt to perform. This process creates the reactive behaviour desired of the system. Each action is selected using information about the current situation.

The best rule for a particular situation is found using two steps. First, each rule is compared to the current situation and a score is calculated measuring how appropriate the rule is for the current situation. Second, the scores of all the rules are compared and sorted, and the rule with the best score is chosen.

4.2.1 Comparing a rule to the current situation

For a single rule, the system calculates a score for the rule by comparing the rule state to the current state of the world. This score is a measure of the similarity of the rule state and the current state of the world. The algorithm for this procedure is as follows:

```
For each object in the rule state
     Find the object in the current state that is most
      like this one
     Calculate a score for this matching
```

```
Sum the scores for each of the object matches,
Normalise the sum by the number of objects in the
 rule state.
Return this value as the score for the rule.
```

Finding the object in the current state that best matches an object in the rule state uses the same principle of scoring used for the rules: match the object in the rule state to every object in the current state, and choose the best score. For each object matching, a score is calculated to represent how good the match is. The score of a match is a modified dissimilarity measure: the details of the two objects are examined; wherever there is something *missing* in the object from the current state that is present in the object from the rule state, increase the count against the score.

Something *missing* means that either the object from the rule contained an attribute that the object from current state did not contain, or the two objects contained the same attribute but the values of the attributes were incompatible. Incompatible values for attributes occur when none of the set of values from the attribute of the rule state object are represented in the set of values in the attribute of the current state object. The use of *any* in the rule state complicates this score: if the value of an attribute of an object in the rule state is *any* , then it will match with all possible values for that attribute, as long as that attribute is present in the current state object (the value of that attribute need not be examined).

The second addition to the rule state also affects the scoring. If an attribute in the rule state is labelled 'must have', and the current state object does not possess that attribute, then the score for that match is automatically set to the maximum (worst) score and the matching process aborts. Similarly, an attribute tagged as 'must not have' in the rule state object that is present in the current state object will cause the matching to fail if the current state object has that attribute.

Particular weight is given to the type attribute of objects. All attributes of an object can be changed during the simulation except the type of the object – if an object is a cup, then it will always remain a cup. The type of the object can not be altered, so if the rule stipulates a cup is to be used, the agent should most probably try to use a cup, rather than a table. Such rules used in the real world would follow a similar form. Rules would identify an object category, for example a drinking vessel, which could be cup, a glass, a mug, etc, and would work correctly as long as the object used is of that type. Using a plate instead would probably not be so successfully. Craneworld does not currently include an object class hierarchy, so the type of objects in a rule is simply the most specific object category. To weight the type of objects above other differences, a mismatched type counts as ten standard attribute differences. The effect of this is that objects will virtually always be matched to an object of the same type, if there is one present, before being matched to objects of other types.

Once each matching of objects has been scored, the matchings to use for the rule are chosen. The matchings chosen are the ones with the best score for each object in the rule state. The score for the rule is then calculated by summing the scores for the matchings and normalising. The current normalisation employed simply divides the sum by the number of objects in the rule. This is an attempt to adjust the difference scores to account for more objects in the rule - the more objects there are, the higher the possibility for differences. Or put another way, the more objects there are, the better large similarities between the rule state and the current state. A rule state with many objects is more specific than a rule state with only a couple of objects. Minor differences in the specific rule are probably less important than the same differences in the general rule.

4.2.2 Finding the best rule from the collection

With the score for each rule normalised, the best rule from the collection is identified by taking the scores for all the rules and selecting the best (lowest) one. The best rule will indicate the action the agent should perform.

4.2.3 Generating the action

The action finally recommended by the rule is generated by using the action argument tags. For each object in the rule state that has an action argument tag, the system replaces the rule state object with its matched current state object. So the action returned will refer to current state objects rather than rule state objects.

4.3 Creating Goal-directed behaviour

The goal is included in the rule so that the actions selected by the RPLA system will change as the goal changes. However, the algorithm described above does not take any notice of the goal, and simply performs matching based on the rule state and the current state. This section describes how goal-directed behaviour is incorporated into the system by modifying the matching algorithm so that the behaviour is dependent on the current goal.

Satisfying a goal will require a number of actions to be performed on objects in the world, where the purpose of the actions is to modify the objects in some way so that they satisfy the requirements of the goal. The RPLA system does not attempt to figure out the purpose of an individual action, but records the goal along with actions that are performed to satisfy the goal. When the agent is presented with a new task to perform, it assumes that it has already been shown all the actions that it needs to be able to satisfy that goal. For example, if the goal requires "Object A contains Object B", two actions that are important for this goal are picking up Object B and putting Object B in Object A. For the system to be able to accomplish any tasks that require one object being inside another, it must first be shown these two actions. Once these actions have been shown, the system should be able to transfer this knowledge to any tasks that require one object be in another. This is one of the fundamental principles of the RPLA system: knowledge gained from experience in the world can be used to satisfy new goals.

To achieve goal-directed behaviour, the system must match objects using more than their description similarities. The matching procedure must consider the objects in relation to the current goal. The actions performed on objects are dependent on the requirements of the goal, not on the particular details of the object description. For example, examine the goals shown in Figure 4.2. The Coffee object in the first goal and the Tea object in the second goal are functionally equivalent, since both objects are required to be in the cup. Satisfying the tea goal will use the same actions as the coffee goal; however the actions will be performed on a Tea object rather than a coffee object.

Implementing this reasoning in the system uses a three-stage matching process that finds objects with equivalent roles in the current state and overrides the basic matching algorithm to only examine these objects. See Figure 4.3.

The first stage is to match objects in the goal with objects in the state of the world. The details of how this matching is performed will be described in fully in section 5.1.6. For now, it is enough to know that the objects in the goal are matched with the appropriate objects in the

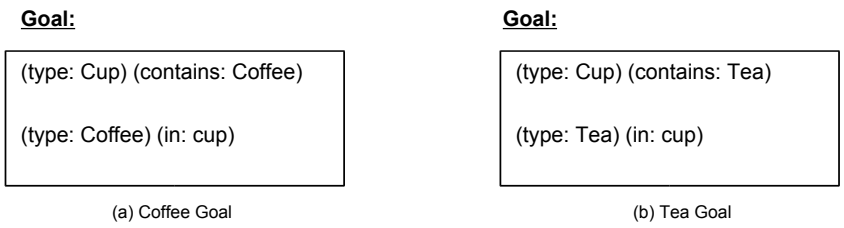
rule state – the objects that will become the goal-satisfying objects. So if the goal is as shown in Figure 4.2.a, the *coffee* and *cup* objects in the goal will be matched to *coffee* and *cup* objects in the state of the world and will have appropriate tags added. The objects in the rule state have already had these tags added. The effect is that when a rule is compared to the current situation, both the rule state and the current state have tags on objects that correspond to objects in their respective goals.

The second stage matches the requirements of the two goals. The requirements of the goal are the description of the objects given in the goal. To satisfy the goal, all the requirements must be met, so, unless already satisfied, every requirement in the goal will require actions to be performed on the world.

Two objects in the goals are matched if both objects have a requirement in common. Note that an object in a goal could match to different objects for each requirement of that object. All matchings are recorded to be evaluated later.

The final stage takes the work from the previous two stages. The algorithm described in the previous section matched all objects in the rule state to all objects in the current state. The final stage for creating goal-directed behaviour overrides this algorithm so that, for tagged objects, only relevant objects are matched and scored. If an object from the rule state is tagged to a goal object, then the system finds the corresponding object in the rule goal, and then finds all the objects in the current goal that were matched to this object in the second stage. The system then finds the objects in the current state that were tagged to these objects in the first stage. Finally, the objects from the current state found by this process are matched to the original object from the rule state. One last consideration is made to facilitate the matching process: when the objects to be matched are found in this manner, a mismatch of type does not incur the same penalty as in a normal matching – in fact it is not penalized at all. The justification for this is that when the matching is dictated by the goal requirements, it is quite likely that the types of the objects that should be matched will be different. If we are matching objects based on how they will be used, rather than simply the object descriptions, then the functional considerations override the normal requirement for identical types.

Figure 4.3 illustrates this process. In the first stage, *cup* and *coffee* in the rule goal are matched to *cup* and *coffee* in the rule state, as are *cup* and *tea* in the current goal matched to *cup* and *tea* in the current state. The second stage matches *cup* in the rule goal with *cup* in the current goal (both are required to contain something) and *coffee* in the rule goal to *tea* in the current goal (both are required to be in something). The final stage will consider the *coffee* object in the rule state, follow the tag to *coffee* in the rule goal, find that it is matched to *tea* in the current goal, then see that *tea* is matched to *tea* in the current state, and so will match the *coffee* in the rule state to the *tea* in the current state and calculate the score for this match, ignoring the different types. Notice that the *coffee* in the current state is not matched to *coffee* in the rule state. This is crucial for the operation to succeed. If the two *coffees* (*coffee* in rule state and *coffee* in current state) were matched, then this matching would almost certainly receive a better score, thus ruining the attempt to direct the actions based on the goal. Following the same procedure, the *cup* in the rule state will be matched to the *cup* in the current state. The best match to *coffee* in the rule state will be the *tea*. Since the *coffee* in the rule state is the argument of the action in the rule and it is matched to *tea* in the current state, the system will substitute *tea* for its *coffee* and the action returned will be ”Pick up Tea”.



(a) Coffee Goal

(b) Tea Goal

Figure 4.2: Two Goals

4.4 Feedback from the teacher

Every action selected by the agent is confirmed or rejected by the teacher before the agent performs that action. If the choice made by the system is the right one, the teacher will confirm it and the agent will perform the action, completing the iteration. If, however, the action is the wrong one, the teacher will reject the choice and the system will have to go back and select a different action.

Supervision from the teacher is needed while the agent is still learning from experience. The agent is choosing actions using the rules that it has created from previous examples. These rules were created using a minimal amount of experience, so it is quite likely that they will not cater for all the scenarios to which they should or should not apply. Getting the rule perfect will require seeing every possible different form of the examples, as well as a number of 'near-miss' examples to show what shouldn't be included. Rather than requiring the agent to experience all these examples before constructing the rules, the RPLA system uses a single example to base the rule on. Because the rules will not be perfect when first created, the matching process cannot take a strict matching approach (either applies or does not apply), but must allow partial matches and grade the rule using the scoring system described above. Then the rule the system selects is the one with the best score from a collection of rules that might all have only slightly worse scores. There is no guarantee that the *right* rule will be the one with the best score.[1]. To ensure that the agent always performs the correct action, and to provide feedback on the accuracy of the rules, the teacher confirms or rejects each action choice by the RPLA system.

If the agent selects the correct action, the right rule was selected to apply. The RPLA system has managed to sift through the rules and come out with the right answer. The teacher accepts the action, indicating to the system that the choice was correct, and the agent performs the action, thus ending the iteration.

If the action first selected is incorrect the teacher will reject the action choice and the system must try again. A simple approach would be to use the next ranked action, and continue until the teacher accepts a choice. However, it is possible that the first rule was the correct rule, but the agent found the wrong correspondence between objects, and so recommended the action with wrong arguments. Choosing an alternate rule would not help. Therefore, when the teacher rejects an action, RPLA must consider other possible matches using the same rule, and rank each of these matchings against the previously found ones.

This requires an extension to the matching process so that RPLA can find multiple possible

[1]Once the rules are perfect the right rule will always have a perfect score, but this does not happen while the rules are still being learnt

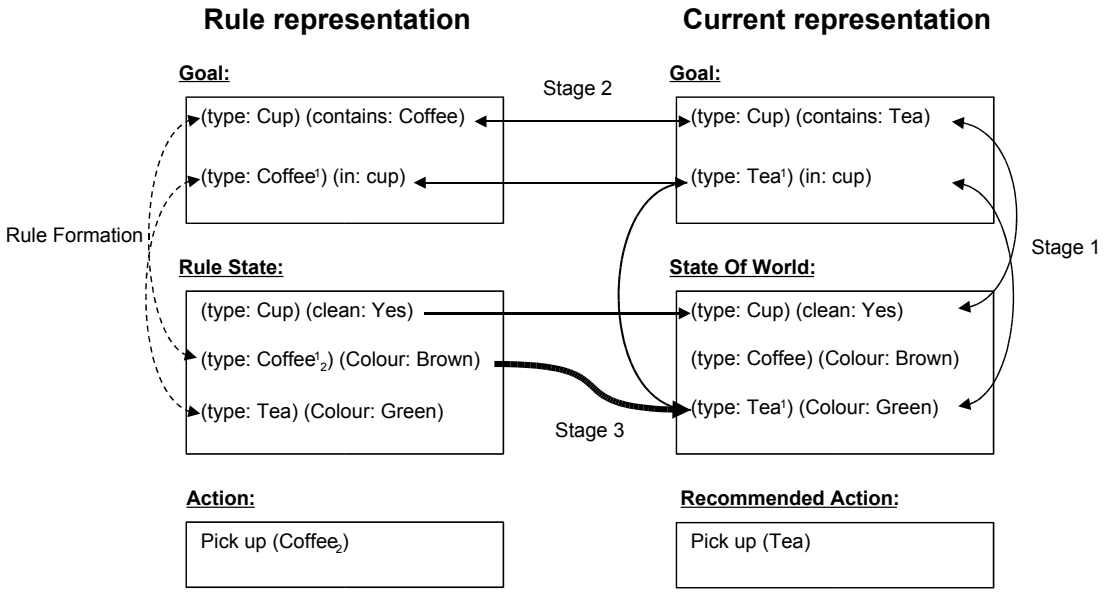


Figure 4.3: Using the Goal to direct the Match

matchings between a rule and the current state, without considering an exponential number of useless matchings.

Chapter 5

Creating the Rules

The ability of the agent to translate its experience into rules to govern its behaviour is the primary learning function of the agent and, combined with the generalisation and specialisation techniques described in the following chapter, is the way the agent learns how to accomplish new tasks.

The agent learns from experience, so to teach the agent a new task, the agent is simply put in the world and shown how to accomplish it. This can take the form of the agent merely observing while the teacher chooses the actions to perform or the teacher confirming and denying the actions the agent selects as described in the previous chapter. In either case the agent will be told which action is the right one to perform, so the agent is receiving new information that it must incorporate into its knowledge base.

The next section describes the process by which the agent constructs a new rule. Chapter 6 describes how the agent modifies existing rules to keep them consistent with its experience.

5.1 The Information Provided

A rule is constructed from a single step of the simulation, so the RPLA system must form the rule using only the information contained in that step. As stated in Chapter 3 a step of the simulation is comprised of the goal, the before-state, the action and the after-state. Figure 5.1 shows all information the system is given to construct a rule that can be used in the future.

5.1.1 Making use of the information

The agent must take the information provided from the step and construct the rule. The rule will contain a representation of the state of the world and the goal that will be matched to the current state and the current goal when the rule is applied. The simplest approach to constructing the rule would be to take the entire before-state from the step and set this as the state of the world in the rule. When given the exact same situation again the rule is guaranteed to apply correctly; however it is unlikely to be very useful in other situations because it contains information about extraneous and irrelevant objects that will interfere with the scoring system.

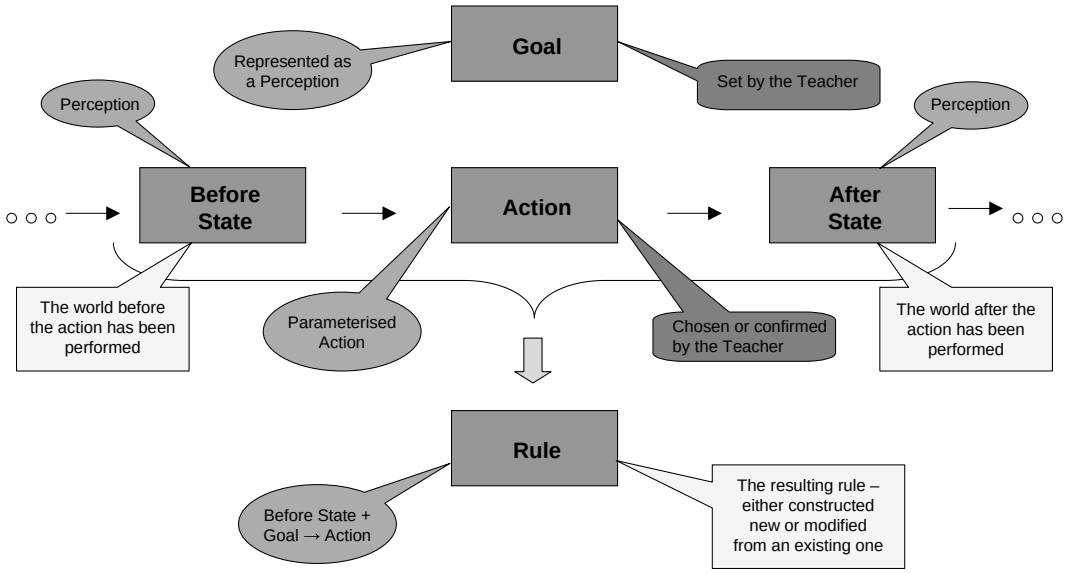


Figure 5.1: Using one Step to create a Rule

5.1.2 Focus of Attention

Rather than using the entire before-state, the state of the world recorded in the rule state is reduced to a much smaller subset of the before-state, containing only those objects that are necessary for the rule. This makes the rule more likely to match properly in the future when it is applied to new situations.

Using focus of attention heuristics is a useful technique for reducing a large collection of objects from a world description to a small, concise set containing only the objects that we really need. Heuristics similar to the ones used by the RPLA system are used in [9]. An object is considered important for the rule if it:

- plays a part in the action, or
- is a goal-satisfying object

These two criteria are used to provide the rule information that is required to correctly determine when the rule should be applied, without the additional objects that are irrelevant for the task.

5.1.3 Inferring Intent

The purpose of the rules is to indicate, in the absence of the teacher, when a particular action should be performed. The rules would perform best if they used the same logic as the

teacher to assess the current situation. If the agent could determine *why* the teacher selected a particular action, it could construct a rule that contains the same reasoning. Unfortunately, discovering intent is notoriously difficult. Particularly with only one example, the agent can not conclude anything reliably. To have any chance of accurately inferring the intent behind the teacher's decisions, the agent would need to collect a group of steps all relating to the same idea (all based on the same reasoning). However we require that the agent learns from a minimal amount of experience (i.e. one step), therefore the agent does not have the luxury of sitting back and waiting for enough examples to be shown to make the details of the desired rule obvious; the rule must be constructed from the information the agent is given from one step.

5.1.4 Action selection focuses on a few objects

Since the system can not determine why exactly an action was chosen, the best it can do is guess the information that is probably relevant for the selection and hope that this will enable the right action to be chosen when system is making the decisions. It should not record the entire before-state because this would be too specific to a particular situation that is unlikely to recur. and recording no information about the before-state would provide no guidance as to when the action should be performed. Therefore the system must use some mechanism to extract a subset of the objects from the before-state to put in the rule state[1]. The focus of attention heuristics are used as an attempt to mimic the technique a person would employ to identify important objects from a large collection when solving tasks.

Consider the general format of the problem: we have a description of the current state of the world, giving details of every object visible, and we have a goal we are trying to reach; the problem is to select the next action to perform. The first thing to do is to examine the goal, determine the requirements of the objects in the goal and match these objects to objects in the current state. That is, the first step is to decide the goal-satisfying objects. Once we have in mind the objects that will be used, we simply choose the actions that will bring us closer to the goal. All the actions we perform will, either directly or indirectly, have some significance to the goal we are trying to achieve; otherwise there would be no point in performing the action. The actions are selected based on their effect on the goal-satisfying objects. Beyond these objects, the only objects in the world that are considered are objects that are needed to perform those actions. All other objects will be ignored. The focus of attention heuristics are based on this theory. From the complete description of the world, the only objects used in the rule state are those identified as potential goal-satisfying objects and those affected by the action performed in that step.

The rule constructed in this way is unlikely to perfect, but it is a useful starting point that allows the agent to make use of the little experience it has received and possibly satisfy a new goal. The rule can then be adapted as the agent is shown more examples.

5.1.5 Identifying the objects used in the Action

Objects that are involved in an action fall into two categories: objects that are arguments of the action and objects that are affected by the action. Typically there will be substantial overlap between these two categories (objects that are arguments are often modified by the action). The objects in both categories can be identified easily in Craneworld. The objects

[1] Even better would be to consider only a small number of the attributes of this subset of objects, but we have no heuristics to determine which attributes are relevant.

that are action arguments are given as part of the action and the objects modified by the action are found by a comparison of the before-state and the after-state. This comparison is very similar to the matching algorithm presented in section 4.2, except before-state and after-state are being compared rather than before-state and rule state, and the process must also identify objects that have been added or deleted in the course of the action. This algorithm was provided as part of Craneworld.

5.1.6 Identifying the goal-satisfying objects

Determining the goal-satisfying objects is more complicated. When the goal is achieved, when the goal-satisfying objects actually play the role for which they are named, the goal-satisfying objects are obvious - these are the objects whose attributes match with all the requirements of some object in the goal. These objects must be found in order to conclude that the goal has, in fact, been achieved. While the goal is not complete, a goal-satisfying object could be in any state between fully meeting the requirements of the goal object and having nothing in common with it except the type, which, of course, will always match. This makes it difficult for the system to determine the goal-satisfying objects. The only thing that can be said about them for certain is that they will be the right type. Since the system can not determine the goal-satisfying objects with certainty, some mechanism must be used to identify these objects as best as possible. A sensible person would choose to use the objects that are closest to meeting the requirements of the goal in their current form, since this means less work will be needed to get to the goal. To achieve this, the same matching process described in section 4.2 could be used. However, it is possible that the best-matching object will not be the one used to satisfy the goal in the end. Even though superficially it appears to be closer to the goal object – meet the requirements more – modifying the attributes of the object so that they correspond with the goal object may actually require more actions than getting some other object in the world to satisfy the goal, and that other object is the one that has been selected by the teacher (who, of course, is intelligent enough to notice this fact.) It is also possible that the goal-satisfying object for some goal object is not present in the current state, for example it is in a different room.

So instead of finding the best match to a goal object, the system identifies goal-satisfying objects by finding all objects that are same type as the goal object. The type is used because it is the only attribute of the object that can be relied on.

Finding all objects with matching type implies the system can find multiple goal-satisfying objects for one goal object, when in reality only one object can be the *actual* goal-satisfying object for that goal object. This could be allowed and the rules would still work, since when the rule is compared to the current state the best match would be found for each of the objects in the rule, however the scoring of the rules would be affected adversely. To overcome this, identifying goal-satisfying objects are only found from the set of objects already determined to be important because they play some part in the action performed.

5.2 Converting a step to a Rule

The process of creating a rule from the information in the step uses three steps:

- Set the Goal in the step as the Goal in the rule
- Set the Action in the step as the Action in the rule

- Extract the important objects from the before-state using the focus of attention heuristics and set this subset of objects as the rule state.

Figures 5.2 and 5.3 show this process for an example step. Figure 5.2 shows the information given to the system as the step, and the objects identified using the heuristics. 'Table', 'Robot' 'Coffee' and 'Kitchen' are found to have attributes modified by the action; 'Coffee' is the argument of the action and is also identified as a potential goal-satisfying object, as is 'Cup'. Figure 5.3 shows the rule created from the step.

5.3 The Effect of Creating Rules from a single Step

For each step performed, the agent must construct a rule based solely on the information from one step. This is a design decision that has some important implications.

Each rule is constructed using just the information from one step, therefore the rules constructed do not rely on the order of the steps. By not considering sequences of steps, the complexity of the system is reduced - two rules constructed from only information about the current step will be identical if the two steps are identical. Considering previous or future steps would result in different rules being constructed depending on the sequence of the steps performed.

Reducing the complexity of the system does, of course, also reduce its capabilities. Each rule is independent; therefore it is much harder to enforce strict sequences of actions. The order in which actions are performed is determined solely by the 'best' rule for the current situation, thus actions can be performed in a different order than when they were shown. This is not a problem for most situations, however it is important that the ordering still results in a consistent set of actions - the agent should not try and put down an object before it has picked it up. Situations that require a more strict ordering - one action must be performed directly after another - can not be represented precisely using this structure. Whether or not the agent can successful perform such tasks will depend on the particular instance for which it is applied, and on the other rules the agent possesses.

Because the rules are constructed solely from the present situation, they are naturally suited to be used in the reactive manner for which they were designed. The agent only uses the information from the current step to construct the rule, so when the agent is selecting actions and only has the current situation as evidence, the rules will lend themselves easily to this task.

By not looking at previous actions the agent is free to do pointless things that could easily be avoided if some history were examined. A simple example is undoing the last action: If the agent looked at the last action that was performed, it could immediately discount any rule that stipulates the agent does the reverse of the previous action.

Figure 5.2: Using the Heuristics

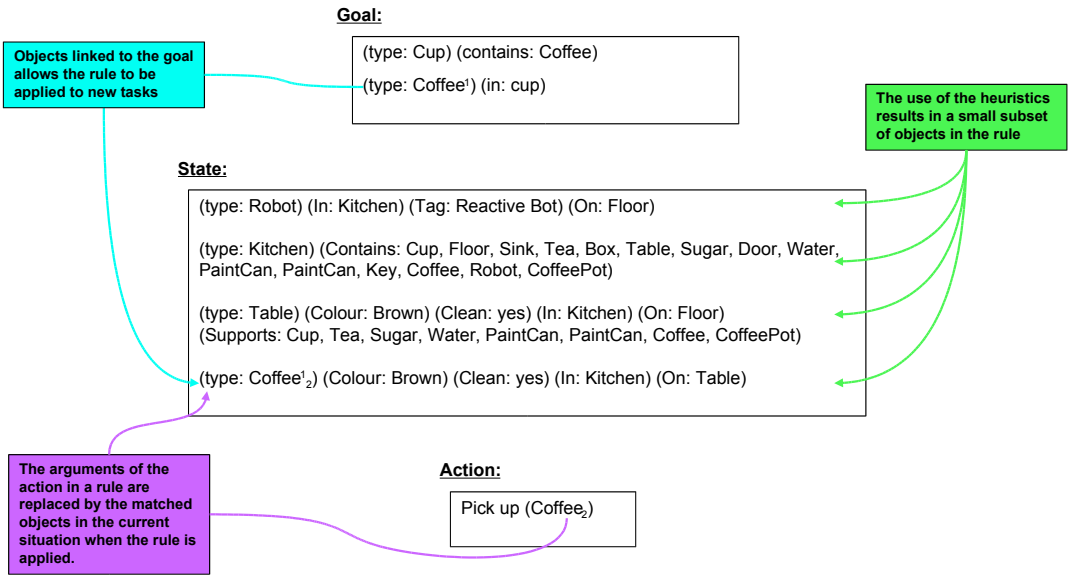


Figure 5.3: The Resulting Rule

Chapter 6

Modifying the Rules

The RLPA system is required to construct rules from single examples to minimise the amount of training necessary. However, when more evidence becomes available – when the system is shown more examples – the system should be able to incorporate this new evidence into the rules it has constructed. With only one example, the accuracy of rules is limited to the efforts made when the rules were constructed, namely the focus of attention heuristics. By continually adjusting the rules to remain consistent with all the experience, the accuracy of the rules becomes limited only by the number of examples used.

The rules are modified after construction using two techniques, generalisation and specialisation. Section 6.1 describes the generalisation process and Section 6.2 describes the specialisation process.

6.1 Generalising the Rules

Generalisation improves the rules constructed by the system when more evidence is made available, allowing specific details about the conditions of the rule to be inferred. When a new example of when a rule should be applied is presented, the system combines this information with that from the previous examples for that rule and formulates a new rule that incorporates all this knowledge.

6.1.1 Why is generalisation needed?

When the rule is constructed, the RLPA system does not have enough information to infer the exact details about which attributes of the objects are important, so, rather than making guesses, the agent reduces the rules state to the full description of each object identified using the focus of attention heuristics, and does not try to look deeper into the problem.

The heuristics are a good starting point for creating rules, and significantly reduce the number of objects considered for matching, but the rule state will still contain the complete description of those objects as they were in the before-state. Much of this information will be specific to the example used to create the rule, and will not relate to the purpose of the rule. For example, if the purpose is to be holding a clean cup, then the fact that the cup is white is irrelevant. This level of detail can not be inferred from a single example, however as more examples are provided that use the same rule, the RPLA system can combine what

it has seen and infer some of this detail based on the attributes that are consistent across the examples.

6.1.2 When is generalisation performed?

Whenever RPLA uses a rule to select the correct action to perform, the system is being provided with a fresh example situation for that rule. The teacher is always monitoring the choices of the system, so the agent selecting the correct action using a rule is providing the same information as if the teacher had selected the action. However, the agent will now have at least one other example to compare with – the current situation and the situation used to construct the rule, providing a means to make comparisons and eliminate attributes. Therefore one rule (the one that recommends the action accepted by the teacher) can be generalised after every iteration of the simulation when RLPA is being used to choose the actions.

6.1.3 How are the rules generalised?

Each time a rule is successfully applied to a situation, the rule is generalised by comparing the rule state with the current state and constructing a more general description of the rule state that includes all the details that are the same in the two states and excludes all those that are different. Because this is done with every new example, and the rule state is always made more general, the resulting rule will always be consistent with all previous examples.

Generalisation of the rule state is performed at three levels. All three types of generalisation are illustrated in Figure 6.1.

- Attribute values are removed: when an attribute is present in both the rule state and the current state, and the values of the attribute in the two states have some parts in common and some different, the value of that attribute in the new rule state will contain only the parts that are present in both states. In the example, the *Table* object contains the attribute *supports* in both the rule state and the current state. In the rule state the value of this attribute is *(Coffee, Tea, Sugar)*, and in the current state the value is *(Coffee, Tea, Box)*. The inconsistencies are removed from the attribute in the new rule state, making the attribute *supports: (Coffee, Tea)*

- Attribute values are made *any*: when an attribute is present in both the rule state and the current state, but the values of the attribute have nothing in common, the value of that attribute in the new rule state is set to *any*. In the example, the rule state and the current state both contain the attribute *colour* of the *Coffee* object. The value of this attribute in the rule state is *Brown*, while the value in the current state is *Black*. Removing the inconsistent values leaves an empty set, so the value of the attribute in the new rule state is set to any to indicate that the attribute is present, but the value of that attribute is considered unimportant.

- Entire Attributes are removed: when an attribute is not present at all in the current state, the attribute is removed from the new rule state. This is shown in the example by the *contains* attribute of *Robot*. Although *Robot* in the rule state has this attribute, *Robot* in the current state does not, therefore the new rule state does not include this attribute.

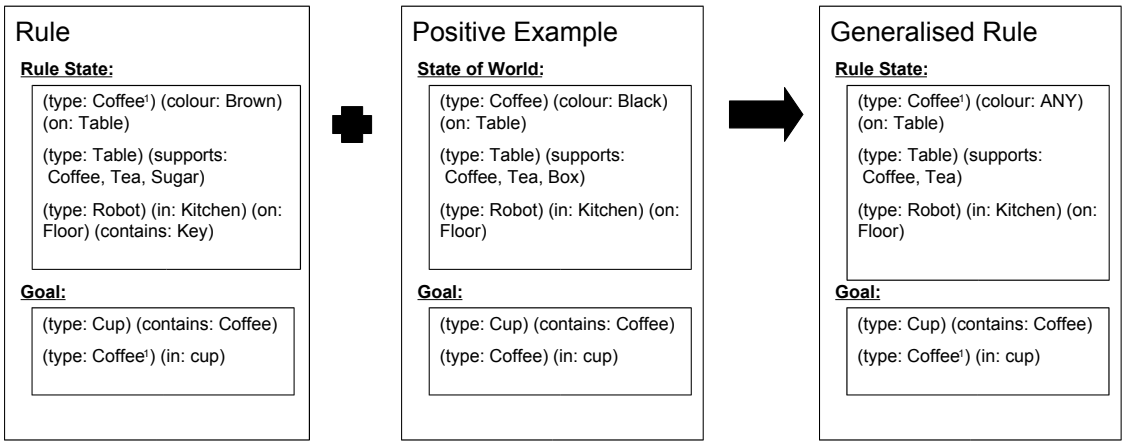


Figure 6.1: Example of Generalisation

6.1.4 Generalising type

Any of the attributes of an object can be generalised, including the type. If the best match to an object is an object with a different type, and the action produced from this is confirmed by the teacher to be correct, then that matching is obviously valid, hence the conditions in the rule should be generalised. In this situation, the type of the object in the new rule state would be changed to *any*. This is not allowed for objects in the rule state that are tagged to goal-objects. These objects are frequently matched to objects of a different type due to the matching algorithm described in Section 4.3. The types of these objects are assumed to be important because of the connection between the objects and the goal objects, so the types are not allowed to be generalised.

6.1.5 Disjunctive conditions are not allowed

Faced with conflicting attribute values, the system removes both sets of values and sets the value in the new rule state to *any*. This is shown in the example above with *colour: Brown* and *colour: Black*. RLPA does not allow disjunctions for the value of an attribute. This greatly simplifies the generalisation process, since the new rule state will either be the same or 'higher up' the generalisation tree rather than moving sideways. It also results in more concise rules, but it could potentially lead to some situations being misrepresented. With this mechanism, a rule can not take the form of "must be (A or B)". If the rule state has 'A', and the current state has 'B', the *right* condition might be 'A or B', but the system will generalise this to *any*. To represent this idea, two rules would need to be created, one with "must be A", and one with "must be B". The system can not determine when a disjunction is required, and so creating separate rules requires that the teacher explicitly show that a new rule is required.

6.2 Specialising the Rules

As with generalisation, specialisation is an attempt to improve the knowledge contained in the rule as more evidence is provided about when the rule should be applied. Unlike generalisation, specialisation occurs when negative examples are presented – when the system chooses the wrong rule to apply to a situation. The rule is modified to prevent it being chosen again in the same situation.

6.2.1 Why is specialisation needed?

Specialisation of the rules is required because of the matching algorithm that allows loose matching of the conditions in the rule. All the attributes given in the rule state are assumed to be flexible. Each difference between the current state and the rule state detracts from the score of the object, but does not prevent the rule from being applied to that situation. If a rule has the best score out of all the rules, then it will be used to recommend the action to perform, even though there may be a number of differences between the rule state and the current state. This is necessary so that the rules the agent has constructed can be applied to new situations. However, not all differences in state can be forgiven as easily: some attributes are significant for the rule to be appropriate. For example, to be able to put something down, the robot must first be holding it. The attribute representing the requirement that the object is in the robot's hand will be present in the rule state, but the matching process will take this attribute to be just another detail, no more important than all the others, and will not reject the rule if the current state does not contain this attribute.

As a result, the RPLA system may use a rule to select actions that are inappropriate for the situation. Specialising the rules improves the accuracy of the system – it makes fewer mistakes – and allows for finer control over the specifics of a rule.

6.2.2 When is specialisation performed?

Specialisation is another way of improving the rules in light of more evidence, so the basic idea is to specialise whenever a negative example is seen – whenever a rule is *not* the right one to apply for a situation. In a sense, every step is a negative example for all the rules other than the one that is actually applied, but this does not necessarily mean that there is anything that needs to be modified about these rules. If a rule is not chosen, the system has no information about whether the rule should or should not have been applied, only that the rule chosen was a better fit to the situation (and was a good rule to use). It might be that several rules are all applicable to a situation and any one could be successfully selected. It is only when the teacher explicitly rejects an action choice that RPLA knows a rule has been used incorrectly. Therefore the system does not attempt any specialisation unless an action choice is rejected by the teacher.

Given that the teacher has rejected the choice of the system, RPLA must decide what to do to the rules to prevent it making the same mistake again. The mistake will fall into one of three categories. The procedure applied by the system depends on the category. The three possibilities are:

- The system used the correct rule, but chose the wrong mapping of rule state objects to current state objects, or

- The system chose the wrong rule to apply, although the action could have been performed, or
- The action recommended was not a valid action given the current state of the world.

If the right rule was used to recommend the wrong action, there is no need to specialise it as the problem will be addressed when the rule is generalised. An incorrect mapping of objects can be found if those objects matched best given the current details of the objects in the rule state. The best mapping given these details may result in the wrong action being recommended. Once told this mapping is incorrect, the system re-evaluates the rule and produces a different match of objects in the rule state to the objects in the current state, repeating, if necessary, until it has an acceptable match of the objects. These objects produce the right action, so the teacher will confirm the choice and the action will be performed. Confirming the action gives an example of when the rule *should* be applied, so the system will generalise the rule to fit with this new example. Once the rule has been generalised, it will match perfectly with that example, so if the same situation is presented again the correct action will be chosen over the incorrect ones (the match will be better), eliminating the problem.

Choosing the wrong rule to apply is more of a problem, as the rule will not be fixed by the generalisation process. With a large collection of rules to choose from, each rule is measured against the current situation and the chosen rule is the one with the best score. There may be only very subtle differences between the chosen rule and the correct rule, but because of the specifics of the current situation the wrong rule was selected.

When the rule selected is not the correct one, but the action is still possible in the current state, the system again does not modify the rule, and lets the generalisation process resolve the problem. Although it may seem logical to change the incorrectly chosen rule so that it will not be chosen over the right one, this is not required and in fact may degrade the quality of the rule rather than improving it. If the action recommended is possible, then the rule is not necessarily malfunctioning, it is just that in this particular instance another rule should have been chosen. When the right rule is generalised, it will match perfectly with the current situation, so will be chosen over the incorrect ones. Thus there is no need to modify the incorrectly chosen rule.

In the final case, the rule selected recommends an action that the agent can not perform in the current state. The loose matching technique overlooked one of the important preconditions for the action and let the rule be applied. In this case there is definitely something about the current state that is not right for this rule – the basic preconditions of the action are not satisfied in the current state. It does not make sense for the system to be recommending illegal actions, so here the rule is specialised to prevent RPLA choosing this rule again in similar situations.

6.2.3 How are the rules specialised?

To specialise a rule, the system identifies the cause of the rule failure and marks this on the rule state. RPLA will recognise if the cause occurs again and will not choose this rule to apply.

If the action could not be performed, then there must be differences between the rule state and the current state that caused the rule to fail since the action was valid when the rule state was first encountered. The first task is to identify all the differences and reduce these

to the underlying differences in the state of the world. Properties differences (such as colour) are already in this form; however, Relationships have inverses, so the system will pair up a Relationship difference with its inverse. For example, if a cup is on a table in one state, but no on the table in the other state, there will be two differences in the state descriptions – the description of the cup will specify that it is on a table and the description of the table will specify that it is supporting a cup.

6.2.4 Making the task easier

With the differences identified and paired up, the system must figure out why the rule failed. There is most likely only a single reason for the rule failing – one precondition not met. Unfortunately, when there are a number of differences between the rule state and the current state the system has no method of determining which difference was the reason for the failure. Of the differences identified, most will be circumstantial differences due to the particular example being used, rather than something to prevent the rule from working. Faced with this problem, the rule specialisation uses two heuristics to reduce the difficultly of the problem:

- Think positively
- Don't try too hard

An action may be invalid either if the current state contains an attribute that prevents the action, or if the current state is missing an attribute that is required for the action. This corresponds to things that 'must not be' in the state and things that 'must be' in the state respectively. To reduce the complexity of the rules, the system takes a positive approach. If a rule fails, the failure is assumed to be caused by a 'must be' attribute that is not present, rather than a 'must not be' attribute that is present. Taking the positive view fits better with structure of the rules, is much more akin to the human approach to such problems, and has a historical significance to systems created in the past: STRIPS rules [6] only state the conditions that must be satisfied for an action to be performed; they do not include details about what must not be satisfied. Similarly, Winston's learning system preferred to assume that a "near miss" was the result of something missing rather than something being added [18].

The second heuristic for addressing the problem simply recognises that the system will not be able to reliably determine the difference that is at fault if there are too many to choose from, so it should not put in too much effort trying. If the number of differences between the rule state and the current state are too numerous, the system should give up and wait for a more useful example, rather than guess wrongly.

6.2.5 A failed technique

The original approach used by the system considered *too* many differences to be 'more than four'. A rule was specialised by making a copy of the rule for each difference between the two states, and in each copy marking that difference as 'must be'. This covered all possibilities, but meant a number of rules would be created that were not consistent with the concept desired. The system was required to keep track of all the rules that were generated during a specialisation procedure, and continually compare the rules to determine which rule was the right one based on when it did or did not apply. Not only was it very difficult for the system

to ever eliminate the incorrect rules created, requiring this back-tracking mechanism seemed an unrealistic complication to include in the system. Therefore this original approach was abandoned, and a much simpler approach was used.

6.2.6 There can be only one!

Under this new technique the system only performs specialisation if there is a single difference between the two states. If there is only one difference, then the system knows that this must be the cause of the rule failure. This corresponds to Winston's principle of "near misses" [18].

Figure 6.2 shows this process for an example rule. The differences between the rule state and the state of the world in the negative example are:

- *(on: Table)* of *Coffee* in the rule state
- *(supports: Coffee)* of *Table* in the rule state
- *(in:Robot)* of *Coffee* in the current state
- *(contains:Coffee)* of *Robot* in the current state.

After pairing up the relationships with their inverses, this leaves two differences: *Coffee* is on the *Table* in the rule state, and not in the current state, and *Coffee* is in the *Robot* in the current state and not in the rule state. Note that these two things are two different causes, even though they both relate to the position of the *Coffee*. However, removing the negative cause leaves only one difference, *Coffee* on the *Table*.

With the cause identified, the system sets the corresponding attributes to 'must be' in the new rule state.

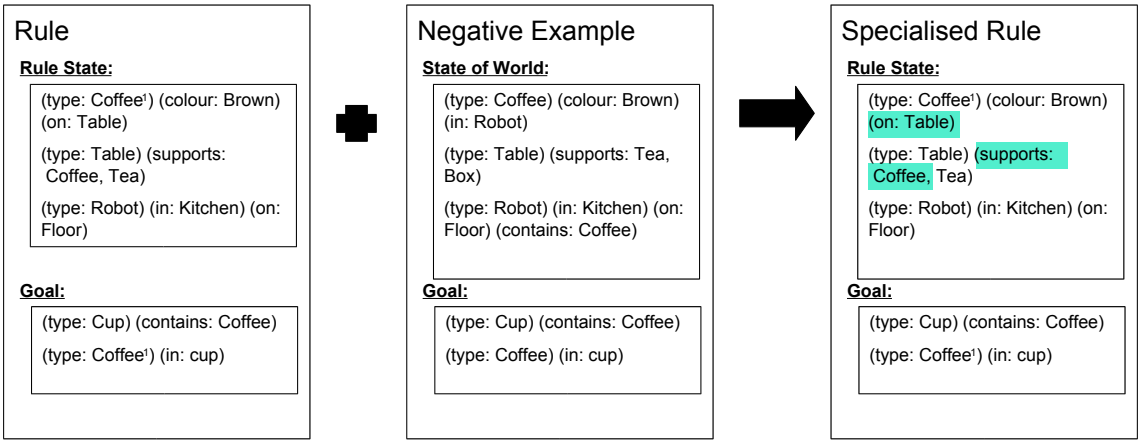


Figure 6.2: Example of Specialisation

6.2.7 Generalise up, Specialise down

Specialisation is essentially the inverse process of generalisation, so it make sense for specialisation to be performed in the reverse order. When generalising, attribute values are first removed one by one, then the attribute value is set to *any*, before the entire attribute is removed. Specialisation goes in the opposite direction. When a rule is first specialised, the attribute identified is set to 'must be'. With this, the rule-matching will require that the attribute is present in the current state, but will not enforce a particular value on the attribute. If the rule is specialised again, and it is the same attribute at fault, the attribute value is set to 'must be'. When this happens, for the current state to match with the rule it must contain the attribute, and the value of that attribute must include the value in the rule state.

Chapter 7

Evaluating the System

The effectiveness of the system is measured by its ability to use the experience it has received to achieve new tasks. To test this, experiments were performed where the agent was provided with an initial goal and shown how to achieve that goal, then given new goals and asked to select the actions to achieve those goals. The tests are designed to assess various aspects of the RPLA system.

Two world configurations were used for the tests. These are shown in Figures 7.1 and 7.2. For each test, the configuration used is *World Configuration 1*, unless otherwise stated.

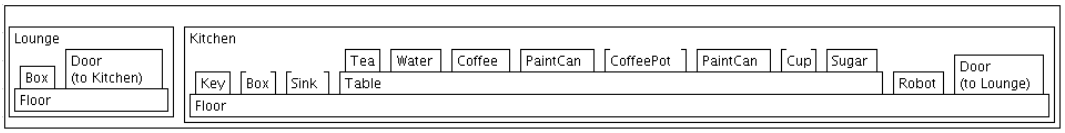


Figure 7.1: World Configuration 1

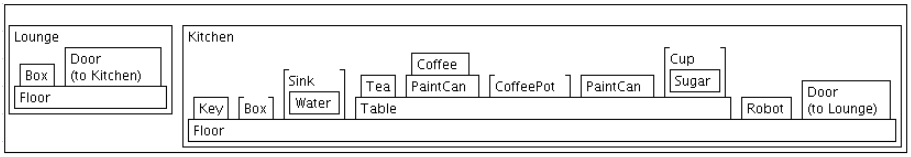


Figure 7.2: World Configuration 2

For the first five tests, the configuration of the world used for training was *World Configuration 1*, and the agent was shown how to achieve the goal shown in Figure 7.3.

To satisfy the goal, the actions selected by the teacher were:

1. Pick up *Coffee*
2. Put *Coffee* in *Cup*

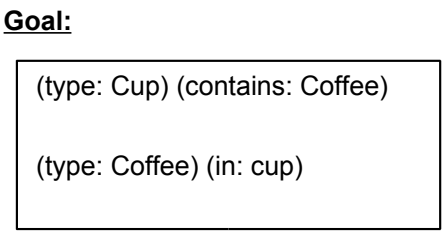


Figure 7.3: Goal 1

7.1 Test 1 – Action Reproduction

The first test is simply to show that RPLA can duplicate the action selections made by the teacher. The agent is shown how to achieve a goal and then given the exact same conditions and asked to select the actions needed. This shows that the agent is capable of creating rules from its experience, and these rules can be used to evaluate the state of the world and produce an action.

The goal being used is *Goal 1*. After being shown how to achieve the goal, the world was set back to the original configuration and the system was asked to satisfy the same goal.

The actions chosen by the agent to satisfy this goal are shown in Table 7.1.

Table 7.1: Actions selected by the Agent for Test 1

Action Chosen	
Pick up *Coffee*	Correct
Put *Coffee* in *Cup*	Correct

7.1.1 Analysis

Reproducing actions given the identical conditions does not test the true purpose of the system, but this basic test is useful to show that the agent has managed to learn rules that it can apply. The agent chose the exact same actions as the teacher, successfully achieving the goal.

7.2 Test 2 – Adapting to changes in the World

The second test uses a new configuration of the world, to show that the agent can use the rules it has learnt when the current state of the world is different from the state when the rules were learnt.

The training demonstrated how to achieve *Goal 1* under *World Configuration 1*. The world was set to a different configuration, *World Configuration 2*, and the agent asked to achieve the same goal.

The actions chosen by the agent are shown in Table 7.2. :

Table 7.2: Actions selected by the Agent for Test 2

Action Chosen	
Pick up *Coffee*	Correct
Put *Coffee* in *Cup*	Correct

7.2.1 Analysis

As in the previous test, the system chose the exact actions that were required. The world was modified from when the rules were learnt, and the agent was still able to choose the right actions to perform. Although the task was very simple, the fact that the right objects were chosen, such as picking up the *Coffee* from the *PaintCan* rather than the table, shows that RPLA is using the matching algorithm to choose the best objects for the rule.

7.3 Test 3 – Adapting to changes in the Goal

The third test demonstrates one of the key objectives of the project. After being shown how to achieve one goal, the agent is provided with a new goal and asked to achieve that using only what it learnt from the previous goal. The major problem for the system in this test is being able to match the objects correctly given the task. That is, RPLA must use the comparison of current goal to rule goal to decide the objects that should be matched in the current state.

To test the goal-matching process, after training had completed the agent was given a new goal, shown in Figure 7.4, but the configuration of the world was the same as the training (*World Configuration 1*). This is important to properly test the goal-directed behaviour. The only change from the setup used for training was the goal, therefore any variation in the behaviour of the agent seen in Test 1 will be due to the new goal.

The actions chosen by the agent are shown in Table 7.3. :

Table 7.3: Actions selected by the Agent for Test 3

Action Chosen	
Pick up *Tea*	Correct
Pick up *Water*	Correct
Put *Tea* in *Cup*	Correct
Put *Water* in *Cup*	Correct

7.3.1 Analysis

This test was a substantial success for the system. As with all the tests seen so far, the agent chose all the right actions without faltering. This test is particularly important because it

Goal:

(type: Cup) (contains: Tea, Water)

(type: Tea) (in: cup)

(type: Water) (in: cup)

Figure 7.4: Goal 2

shows that the actions selected by RPLA were altered to match the requirements of the goal. The state of the world presented to the system was the same one used in Test 1, where the system performed the actions on the *Coffee*. Here, with the goal instead involving the *Tea* and *Water*, the system is able to match the requirements of the new goal with that of the rule goal and generate the right actions to perform. This result demonstrates that the system is capable of using the goal to direct its behaviour. The actions chosen were in a slightly different order than was expected —both 'pick up' actions were performed before the 'put in' actions, however the chosen sequence is just as valid.

7.4 Test 4 – Modifying the Rules Part 1

Both this test and the following one demonstrate the agent modifying the rules that it has previously created as it receives more experience. Combining examples to improve the rules is important so that giving the agent more experience will result in better rules. To provide a good example of this working effectively, two tests are used that show the difference in the actions selected by the system. Two separate runs are required so that the knowledge the agent learns from the first test is not used in the second test.

The system was trained as before, then starting with same configuration – *World Configuration 1*, the system was first asked to satisfy a new goal, shown in Figure 7.5. When this was done, the world was reset and the system was asked to satisfy the goal again. Although Test 1 showed that the agent can repeat action choices, for this test the agent is not constructing new rules for the actions that it chooses, but instead modify pre-existing ones.

The actions chosen by the agent on the first run are shown in Table 7.4. After resetting the world, the actions chosen by the system for the second run are shown in Table 7.5.

:

Goal:

(type: Box) (contains: Cup, Key)

(type: Key) (in: Box)

(type: Cup) (in: Box)

Figure 7.5: Goal 3

Table 7.4: Actions selected by the Agent for Test 4 Run 1

Action Chosen	
Pick up *Cup*	Correct
Pick up *Key*	Correct
Pick up *Cup*	Incorrect
Pick up *Key*	Incorrect
Put *Key* in *Box*	Correct
Put *Cup* in *Box*	Correct

7.4.1 Analysis

On the first run through, the agent selected an invalid action twice. While the robot was already holding the *Cup* and the *Key*, the system decided first that the best action to perform was to pick up the *Cup*, then the next best action was to pick up the *Key*. It wasn't until the third attempt that a good action was selected - putting the *Key* in the *Box*. The two invalid actions were generated by the same rule, using two different object matchings, both of which scored better than valid actions. The actions were both invalid for the same reason – the agent is trying to pick up objects that it is already holding, something the rule doesn't say is illegal. Two invalid actions are chosen (rather than one) because the rule is not modified

Table 7.5: Actions selected by the Agent for Test 4 Run 2

Action Chosen	
Pick up *Cup*	Correct
Pick up *Key*	Correct
Put *Cup* in *Box*	Correct
Put *Key* in *Box*	Correct

until the end of the iteration – after an action has been selected and performed on the world. After the correct action is found, RPLA can attempt to modify the rules to prevent this happening again. The rule used to generate the correct action is generalised, however due to there being too many differences between the rule state and the current state, the rule that suggested the invalid actions is not specialised.

Nevertheless, the generalisation performed is all that is required for RPLA to select the correct actions the next time around. After the system has satisfied the goal once, the world is reset and the agent is made to satisfy the goal again. This time, the agent selects only the right actions, and each time the score of the action selected is perfect. The existing rules have been generalised to cover all the examples seen so far, thus they act just as if this goal had been shown to the system by the teacher.

7.5 Test 5 – Modifying the Rules Part 2

This test follows a similar format to the previous one, showing that RPLA can use experience from different examples to create rules that will work with new examples.

After training, the agent is asked to satisfy *Goal 2*. This is exactly what was shown in Test 3. When this had completed the world was reset and the agent was given *Goal 3* to satisfy. Unlike the previous test, here the agent is testing the generalised rules on an unseen example.

The actions chosen by the agent to satisfy *Goal 2* were the same as in Test 3. For completeness they are shown again here in Table 7.6. This time when the world is reset, the agent is given *Goal 3* and asked to satisfy it. The actions chosen are shown in Table 7.7.

Table 7.6: Actions selected by the Agent for Test 5 *Goal 2*

Action Chosen	
Pick up *Tea*	Correct
Pick up *Water*	Correct
Put *Tea* in *Cup*	Correct
Put *Water* in *Cup*	Correct

Table 7.7: Actions selected by the Agent for Test 5 *Goal 3*

Action Chosen	
Pick up *Cup*	Correct
Pick up *Key*	Correct
Put *Cup* in *Box*	Correct
Put *Key* in *Box*	Correct

7.5.1 Analysis

After the achieving *Goal 2*, the rules are generalised to match with the new situations they were applied to, so when the agent is given *Goal 3* to satisfy, the actions are selected correctly

without any problems. Test 4 was necessary to show that if the rules are not generalised then the agent will select incorrect actions for *Goal 3*. However, with another example to compare the rules to, the details of particular objects in the rule that affected the scoring in the first run of Test 4 are removed and *Goal 3* is satisfied successfully without choosing any invalid actions.

7.6 Test 6 – Combining Goal Requirements

The training process for this test showed the system how to satisfy two goals, rather than just *Goal 1* used by all the previous tests. The agent was shown how to satisfy *Goal 1* given *World Configuration 1*, and then the world was reset and the agent was shown how to satisfy *Goal 4*, shown in Figure 7.6. This test is used to show that the system can take what it has learnt from different tasks and apply what is needed to solve a more complicated task.

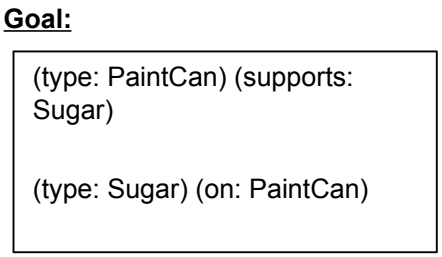


Figure 7.6: Goal 4

To satisfy *Goal 4*, the actions selected by the teacher were:

1. Pick up *Sugar*
2. Put *Sugar* on *PaintCan*

After being shown how to satisfy these two goals, the system is given another goal, shown in Figure 7.7. To achieve this goal RPLA must borrow from the rules created for both the goals it has been taught.

The actions chosen by the agent are shown in Table 7.8.

7.6.1 Analysis

Although RPLA makes some mistakes about the actions to perform, it is able to take experience from the two different goals and apply it appropriately to satisfy the new one. This is another important success for the system, as it shows that in theory RPLA could be used to satisfy increasingly complicated goals as long as the necessary components have been

Goal:

(type: Cup) (supports: Key)

(type: Key) (on: Cup)

(type: CoffeePot) (contains: Tea)

(type: Tea) (in: CoffeePot)

Figure 7.7: Goal 5

Table 7.8: Actions selected by the Agent for Test 6

Action Chosen	
Pick up *Tea*	Correct
Pick up *Key*	Correct
Put *Key* on *Cup*	Correct
Pick up *Key*	Incorrect
Pick up *Tea*	Incorrect
Put *Key* on *Cup*	Incorrect
Put *Tea* in *CoffeePot*	Correct

demonstrated. Incorrect actions were chosen in this test due to the generalisation performed on the rules. After the pick up actions had been performed, the rules were generalised to include those examples. When the system came to select another action, the best score was for actions that it had just generalised, since these had been made to fit perfectly with the previous examples.

7.7 Summary of Results

From the set of tests shown above, the following can be concluded about the abilities of the system:

- The system is able to create rules from its experience
- The rules created can be applied to new situations
- The behaviour of the agent is controlled by the goal primarily, the state of the world secondarily.
- The system generalises the rules as it receives more experience

- Knowledge about two or more different goals can be combined to achieve a new goal.

7.8 Limitations

7.8.1 The goal-matching is too limited

The goal-directed behaviour relies heavily on the relevant objects being specified in the goal. Thus the goal contains both sides of the relationship (such as 'A contains B' and 'B in A') even though satisfying one side will implicitly satisfy the other. Both sides are needed so that the goal-matching algorithm can match up the roles of the objects and select the correct objects to use in the current state.

The goal-matching ensures that the actions selected by the system use the right objects by overriding the standard matching algorithm so that goal-satisfying objects are matched based on their goal requirements. However, the goal can demand different objects to be used in the actions without those objects being mentioned in the goal. If the objects are not in the goal, they will not be tagged as goal-satisfying objects so the normal matching algorithm will be used - choosing the closest match to the object.

The problem this causes the system can be shown with a simple example. First, the teacher shows the agent how to satisfy the goal shown in Figure 7.8. The goal has a simple requirement - a *Door* must be green. Painting the door green requires two actions:

1. Pick up *PaintCan*
2. Paint *Door* with *PaintCan*

There are two *PaintCans* in the world, a green one and a blue one. To paint the *Door* green, the green *PaintCan* must be used.

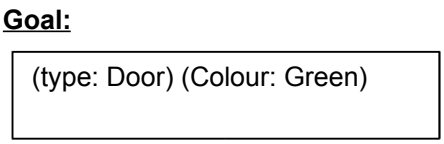


Figure 7.8: Paint Door Goal

Now after being shown this, RPLA is given the same goal, but this time the *Door* must be blue instead of green. The correct set of actions would be as before, except this time picking up the blue *PaintCan*. Unfortunately, the system has no mechanism to tell it that it is the **blue** *PaintCan* that it must pick up not the green one. As illustrated by Figure 7.9, the goal-matching algorithm does not consider *PaintCans* important because they are not goal-satisfying objects, so the normal matching algorithm is used and the best-matching action will be to pick up the **green** *PaintCan* again. After being told that this is incorrect, the next best choice will be to pick up the blue *PaintCan*, and the system will perform this action and generalise the rule to be any *PaintCan*, so it won't prefer green *PaintCans* over blue ones, however it will never learn that to paint something blue it must use the **blue** *PaintCan*. To overcome this problem the goal-matching algorithm needs to perform a deeper analysis of the requirements of the goal and match the objects using more information than just their

type. This was considered earlier in the development of the system but it was decided that it would introduce complication into the system that might not be necessary.

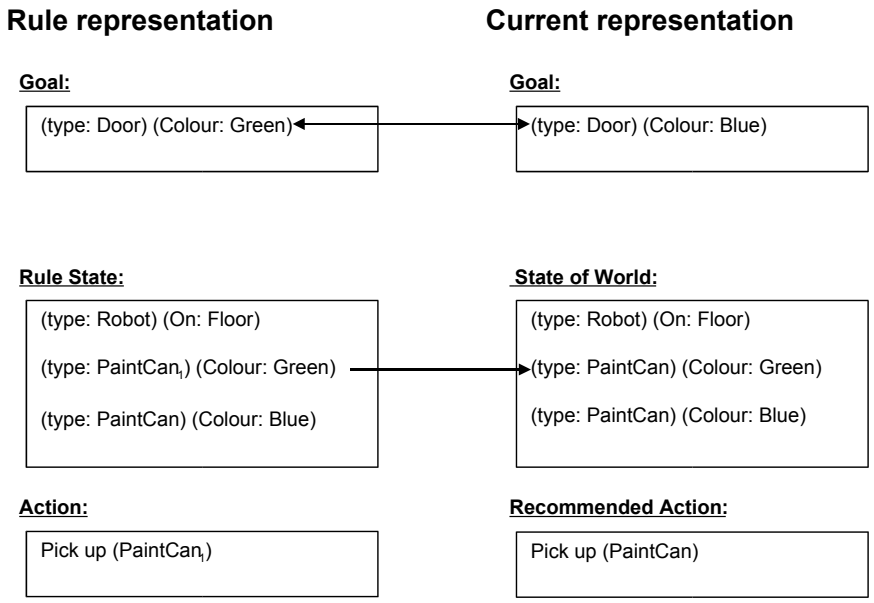


Figure 7.9: Goal-Matching Failing

7.8.2 Specialisation fails miserably

The system is unable to specialise the rules effectively when problems occur. The system can not do anything at all unless there is only a single difference between the rule state and the current state, which obviously only happens when the two states are almost identical. With a multitude of differences, the system does not attempt to identify the cause. To teach RPLA the right specialisation required, negative examples *very* similar to the original examples must be presented that enable the system to reliably determine why the rule failed and prevent it from happening again. A number of examples may be required for a single rule if there are a number of requirements that must be met.

Chapter 8

Conclusions

8.1 Contributions

This report presented the development and implementation of RPLA, a system that learns reactive rules from experience in the world. The rules created by the system incorporate the goal to allow the behaviour of the agent to adapt to meet the requirements of the current objective.

RPLA demonstrates that rules can be learnt from experience in the world, and do not need to be pre-programmed. The system created the rules from examples directed by the teacher. A representation for the rules was developed that could handle the complex nature of the world and scalability was achieved by using focus of attention heuristics to limit the scope of the search performed by the system. With this, RPLA could use the examples presented to construct rules about those examples. The rules are improved after construction by generalising and specialising them when more examples are presented. Thus RPLA is able to improve the accuracy of the rules as it receives more experience.

Reactive behaviour was achieved using two novel matching algorithms to match the rule state with the current state of the world. Without these techniques, the rules would not be usable when the world changed. The object matching algorithm used a 'loose matching' approach that allowed for variations in the state of the world and kept a score of the similarity between the rule state and the current state. Using this score the best action was selected for the current situation.

The goal-matching algorithm matched the requirements of the rule goal with that of the current goal, enabling the system to modify the actions generated by the rules so that they related to the current task. This allowed the system to use knowledge that it had gained from a previous experience to satisfy new, unseen goals.

Heuristics were incorporated into the system to improve the way the rules were specialised. Differences between the rule state and the current state were reduced to their underlying causes by pairing up relationships with their inverses, reducing the number of possible alternative causes.

8.2 Future Work

8.2.1 Combining system with other Agents

An important future direction for this work is to combine the rule learning and reactive rule selection techniques developed in the RPLA system with other learning agents. The problem addressed by this work forms only a small part of the issues faced by agents that learn from the environment. By incorporating a variety of learning and planning algorithms the agent could be made to [autonomously] satisfy highly complex tasks.

One approach to this borrows from the idea used by [14]. The system could learn both reactive rules and more STRIPS-like rules and use both in an integrated forward-chaining planner.

Another technique would be to use a traditional planner as the teacher for a system that learns reactive rules. Given the current state of the world and a goal, the planner sits for a while and comes up with the plan to satisfy the goal. The planner is then used to show the reactive learner the actions that are needed to satisfy that goal. The reactive learner would then create the rules just as before.

8.2.2 Using a more helpful teacher

The direction provided by the teacher to aid the learning process was *minimal*. The only information the system received told it what the correct actions to perform were. The system was required to infer, using heuristics, the details necessary to build the rule. Although adequate for the RPLA system, it is not clear that they will work in more complicated situations. However, the system does not need to rely solely on them when there is a teacher present. If the teacher is going to the effort of showing the system how to achieve a task, then he or she should be able to provide more comprehensive information to the agent about what is going on.

This does, however, come at a cost – the more information the teacher is made to provide, the more taxing the teaching the system new tasks becomes on the teacher. When the teacher has to explain to the agent extra details about the situation, teaching new tasks becomes requires more than just showing the actions required.

One approach that could improve the system without requiring much more effort on the part of the teacher is to identify the goal-satisfying objects right at the beginning. Each time the system is given a new goal, the teacher could indicate the objects that he or she plans to use to satisfy the goal. The teacher chooses these objects before any actions are performed on the world, yet the agent currently has to rediscover these objects after every step. Providing this information to agent requires minimal additional effort from the teacher, but could both simplify the matching process and make it more reliable. With this information at hand, the system could make more complicated inferences about the actions performed by the teacher. In a more complicated world, some actions might be performed by the teacher that did not have any direct effect on anything to do with the goal, but might be necessary to achieve the goal. If the system knows the objects the teacher is focussing on, it can attempt to figure out the purpose of actions such as this.

8.2.3 Delaying Rule construction

The same effect as the above can be achieved without the teacher providing the information by delaying inference. The teacher shows the agent a goal, and then shows it the actions that are required to satisfy that goal. The agent does not know until the end of the process which objects the teacher intends to use as the goal-satisfying objects. However, when the last action is performed (and probably sooner for many of the objects), the objects used becomes obvious. If the system delays the construction of the rules until after the goal has been achieved, it will know the identity of the goal-matching objects when it goes to construct the rules.

8.2.4 Improving the Specialisation

The specialisation techniques proved inadequate for the type of learning that was required. The original approach was abandoned because the system could not reconcile the additional rules that were created. The current approach can learn about some of the complications in rules, but more often than not the system is unable to perform any specialisation due to there being too many differences. More sophisticated techniques could be used that record a collection of negative examples and then attempt to specialise the rule by analysing the collection as a whole. This would require that all the negative examples failed for the same reason, otherwise combining examples would not be of any use.

Another problem is that currently specialisation is used primarily as a way of preventing invalid actions being selected, rather than a means to improve the rules. Altering the specialisation technique so that it is focussed on improving the rules should enhance the effectiveness of the process.

Bibliography

[1] Agre, P.E., Chapman, D.; "Pengi: An implementation of a theory of activity"; In Sixth National Conference on Artificial Intelligence, 1987

[2] Andreae, D.; "Representing, Matching, and Generalising Structural Descriptions of Complex Physical Objects"; PhD Thesis, Victoria University, 1994.

[3] Andreae, J.; "Associative Learning for a Robot Intelligence"; Imperial College Press, London, 1998

[4] Brooks, R.; "A Robust Layered Control System for a Mobile Robot."; MIT A.I. Memo 864. Massachusetts Institute of Technology, 1985

[5] Chapman, D; "Vision, Instruction, And Action"; The MIT Press, Cambridge, Massachusetts, 1991

[6] Fikes, R., N. Nilsson.; "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving"; Artificial Intelligence, 2:189–208, 1971.

[7] Finney, S., Gardiol, N.H., Kaelbling, L.P., Oates, T.; "The Thing That We Tried Didn't Work Very Well: Deictic Representation in Reinforcement Learning"; In 18th International Conference on Uncertainty in Artificial Intelligence (UAI); Edmonton, Alberta, 2002

[8] Firby, J.; "An investigation into reactive planning in complex domains. "; In Proceedings of the 6th National Conf. on AI, Seattle, WA, July 1987.

[9] Foner, L. Maes, P.; "Paying attention to what's important: Using focus of attention to improve unsupervised learning"; In Proceedings of the Third International Conference on Simulation of Adaptive Behavior, Brigton, England, 1994.

[10] Lee, T.; "Exploring Fast Symbolic Learning Strategies in Micro-Worlds"; Honours Project, Victoria University, 2003

[11] McLauchlan, M.; "Reinforcement Learning and Generalisation"; Honours Project, Victoria University, 1999

[12] Muscettola, N., Dorais, G., Fry, C., Levinson, R., and Plaunt, C.; "IDEA: Planning at the core of autonomous reactive agents"; In International NASA Workshop on Planning and Scheduling for Space, 2002.

[13] Russel, S., Norvig, P.; "Artificial Intelligence: A Modern Approach"; Prentice Hall. New Jersey, 1998

[14] Shanahan, M.P.; "Using Reactive Rules to Guide a Forward-Chaining Planner"; Proceedings 6th European Conference on Planning (ECP 01).

[15] Shen, W.M.; "Discovery as autonomous learning from the environment"; Machine Learning, 12, 143-156, 1993.

[16] Sutton, R., Barto, A.; "Reinforcement Learning: An Introduction"; MIT Press, Cambridge, MA, 1998

[17] Williams, B. C., Nayak, P.; "A Reactive Planner for a Model-Based Executive"; In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97), 1178-1185. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

[18] Winston P.; "Learning structural descriptions from examples"; In "The psychology of computer vision."; Edited by Winston P.; McGraw-Hill Book Company, New York, 1975