Representing Qualitative Action Models for Learning in Complex Virtual Worlds

by

Adam Clarke

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington
2011

Abstract

This thesis addresses the problem of representing and learning qualitative models of behaviour in complex virtual worlds. It presents a novel representation, 'Q-Systems', that integrates two existing representation frameworks: qualitative process models and action description languages. Q-Systems combines the expressive power of both frameworks to allow actions and world dynamics to be modelled in a common way using a representation based on non-deterministic and probabilistic finite state machines. The representation supports learning and planning by using a modular approach that partitions world behaviour into 'systems' of objects with specific contexts and a related behaviour.

Q-Systems was developed and tested using an agent in a rich simulated world that was created as part of the thesis. The simulation uses a rigid body physics engine to produce complex realistic interactions between objects. An action system and a qualitative vision system were also developed to allow the agent to observe and act in the simulated world.

The thesis includes a proposed two stage learning process comprising an initial stage in which 'histories' (contextually and temporally restricted sequences of observations) are extracted from interactions with the simulation, and a second stage in which the histories are generalised to create a knowledge base of system models. An algorithm for generating histories is presented and a number of heuristics are implemented and compared. A system for learning generalised models is presented and it is used to assess the suitability of Q-Systems with respect to learning in complex environments.

Planning with Q-Systems is demonstrated in an agent which reasons with generalised models to work out how to achieve goals in the simulated world. A simple planning algorithm is described and a variety of issues are explored. Planning with a single system is shown to be relatively straightforward due to the modular nature of Q-Systems.

This thesis demonstrates that Q-Systems successfully integrate two different representation frameworks and that they can be used in learning and planning in complex environments. The initial results are promising, but further investigation is required to fully understand the advantages and disadvantages of the Q-System approach compared with existing learning systems. This would involve the development of benchmark problems (currently there are none for this particular domain).

Acknowledgments

Many thanks to my supervisor Dr Peter Andreae for his support, ideas, feedback, and enthusiasm.

Contents

1 Introduction **1**

1.1 Goals . . . 2

1.2 Further Chapters . . . 4

2 Related Work **7**

2.1 Representing Actions . . . 7

2.1.1 Actions as Discrete State Change . . . 8

2.1.2 Hierarchies of Actions . . . 12

2.1.3 Actions with Duration . . . 13

2.1.4 Concurrent Actions . . . 16

2.1.5 Delayed and Hidden Effects of Actions . . . 18

2.1.6 Probabilities in Actions . . . 18

2.1.7 Modular Actions . . . 21

2.2 Learning Actions . . . 22

2.2.1 Probabilistic outcomes in a simulated 3D world . . . 24

2.2.2 SLAF: learning in a partially observable world . . . 26

2.2.3 MSDD: probabilistic outcomes with exogenous events 28

2.2.4 EXPO: detecting and refining incomplete actions . . . 30

2.2.5 OBSERVER: learning from plans and experiments . . 31

2.2.6 LIVE: integrating learning and problem solving . . . 32

2.2.7 ARMS: learning from plans with incomplete observations . . . 34

2.3 Representing Qualitative Behaviour . . . 35

2.3.1 Quantities, Constraints and Processes 38
2.3.2 Spatial Representation 41
2.3.3 Actions in Qualitative Systems 43
2.4 Learning Qualitative Models 45
2.4.1 Robotic arm control in a 3D simulated world 45
2.4.2 Predicting qualitative physical interactions 47
2.4.3 QUIN: inducing a qualitative decision tree 48
2.4.4 INTHELEX: learning naive physics 50

3 Q-Systems 53
3.1 Goals 54
3.2 The Q-System Representation 57
3.2.1 Objects 57
3.2.2 States 59
3.2.3 State Transitions 60
3.2.4 Systems 61
3.3 Notation 65
3.4 Design Considerations 66

4 Simulation, Vision System & Skill System 71
4.1 Experimental Setup 75
4.1.1 Software Architecture 77
4.2 Simulation 80
4.2.1 Rigid Bodies 82
4.2.2 Liquids 84
4.2.3 Devices 85
4.3 Vision System 85
4.3.1 Implementing Observables to Support the Q-System Representation 88
4.3.2 Observing Qualitative Shape 91
4.3.3 Observing Qualitative Liquids 93
4.3.4 Observing Qualitative Kinetics 95

4.4 Skill System 96
4.4.1 Example Action Implementation 98
4.5 Test Environments 100
4.5.1 Kitchen World 101
4.5.2 Toy World 102

5 Histories 105
5.1 What is a History 105
5.2 Example Histories 107
5.3 History Generating Algorithm 113
5.3.1 History Generation Heuristics 115
5.4 Effect of Heuristics on History Construction 120
5.4.1 Generating Datasets 121
5.4.2 Results 122
5.4.3 Analysis 125

6 Using Q-Systems in Learning & Planning 131
6.1 Matching 132
6.1.1 System Matching Algorithm 133
6.2 Generalising 135
6.2.1 Model Refinement Algorithm 136
6.2.2 Model Selection Algorithm 137
6.3 A Planner 139
6.4 Learning & Planning in the Simulated World 141
6.4.1 Agent Implementation & Performance 142
6.5 Limitations of the Agent 145

7 Conclusion 149
7.1 Future Work 156

List of Figures

2.1 State transition diagram for a simple STRIPS like action. . . 11

2.2 State transition diagram illustrating conditional (top) and ambiguous action effects. 12

2.3 State transition diagram showing a hierarchy of sequential composite actions. 13

2.4 A durative action modelled as two 'instantaneous' start and stop actions. 15

2.5 A durative action modelled with a fixed duration and intermediate effects. 17

2.6 State transition diagram illustrating probabilistic action outcomes. 20

2.7 A quantity space with 3 landmarks and 5 legal values. 39

2.8 The 'U-tube' system. 39

2.9 Example spatial relations used in the rectangle calculus. . . . 42

2.10 Bayesian network for qualitative state changes. 48

2.11 Example learned qualitative tree. 49

3.1 Part of Behaviour Graph for Simple Sink System. The values for **H**, **F**, and **P** correspond to the variables water-height, flow-rate and plug-in respectively. The sink is empty in states *a* and *c*; the sink contains water in states *d* and *e*; the sink is filling in states *f* and *h*; the sink is overflowing in states *g* and *i*; and the sink is draining in state *b*. 63

4.1 Conceptual Architecture 75
4.2 Software Architecture 78
4.3 A Simulated Kitchen Environment 81
4.4 Approximating Liquid Flow with Particles 84
4.5 Observation Interval 86
4.6 Multiple Transitions Between Snapshots 87
4.7 Hierarchy of Observables 89
4.8 Qualitative Observations of a Simulated Variable 91
4.9 Hierarchy of Shape Classes 93
4.10 Incorrect Calculation of Liquid-Height Derivative 95
4.11 Simultaneous Observation of Dynamic Transition and Action 98
4.12 Kitchen Environment Tap and Sink System 101
4.13 Interactive Toy World Simulation 103

5.1 Behaviour Graph of System Generated from Observing and Interacting with a Rolling Ball. The system has three states: **a**. the ball is stationary; **b**. the ball is moving but its effective speed is zero; **c**. the ball is moving with non-zero speed (see text for explanation). 108
5.2 Observed Tap and Sink Behaviour 111

6.1 Behaviour Graph of a Generalised 2 Block System. The system has four states: **a**. block ?A is falling; **b**. block ?A is partially on block ?B but teetering on the edge; **c**. block ?A is on block ?B, forming a group ?G; **d**. blocks ?A and ?B are on the floor. 143
6.2 Behaviour Graph of a Generalised 3 Block System. 145

List of Algorithms

1 GenerateHistories 114
2 MatchSystems(src, tgt) 134
3 RefineModel($model, history, match$) 137
4 LearnSystems($weights, threshold$) 138
5 AchieveGoal($goal, knowledgeBase$) 140

Chapter 1

Introduction

This thesis addresses the problem of enabling autonomous agents to learn symbolic causal models in complex environments. In particular it presents a new knowledge representation scheme, called ‘Q-Systems’, which builds on existing work in action and qualitative process representation. The representation is intended to allow agents to model and learn about complex real world behaviours. The thesis presents an implementation of Q-Systems in an agent that learns and plans in a richly simulated virtual world.

Creating an autonomous agent that can operate in complex and arbitrary environments is a longstanding goal of artificial intelligence research. Such an agent would be able to learn about its surroundings and use what is has learned to achieve complex goals. This project builds on existing research in this area that focuses on the problems of symbolic knowledge representation and symbolic knowledge learning.[1] An underpinning assumption in these areas of research is that the agent can operate at an abstract level; i.e. the agent observes its world in terms of objects and can manipulate them using a set of pre-learned skills. This is in contrast to approaches using a ‘low-level’ agent, which might observe the world in terms of light sensor readings and manipulating the world via impulses to

[1]Or ’knowledge discovery’ as it is sometimes called.

motors. This assumption is pragmatic when researching high-level learning because there are many difficult low-level issues which are yet to be resolved.

Realistic problem domains exhibit many phenomena that make representing and learning models difficult. These include: the presence of hidden state, inaccurate sensors and actuators, unbounded numbers of objects and varieties of interactions between them, uncertainty in the outcome of actions, and many others. Each of these types of problems have been addressed to some extent by existing action description languages and/or qualitative process representation frameworks. However, there has been very little research into how these representations might be combined and their individual strengths exploited. This project aims to explore how an integrated action description language and qualitative process framework can improve an agent's ability to model complex phenomena and in a way that will enhance an agent's ability to learn such models.

A key part of this work will be to create a simulated world in which the representation will be demonstrated and assessed. Ongoing improvements in computational throughput and new software (e.g. rigid body physics engines) has made the real-time simulation of ever more complex behaviour possible. This project aims to take advantage of these developments to investigate worlds with richer behaviour than those used in earlier investigations.

1.1 Goals

This work has five main goals:

1. Examine the existing action description and qualitative process representational frameworks to determine the strengths and weaknesses of each.

2. Identify the issues and phenomena that make learning in realistic complex worlds difficult.

3. Develop a new integrated representation that expresses *both* actions and qualitative dynamics yet is also amenable to learning.

4. Create an interactive simulated world that exhibits the kinds of phenomena identified, and a simulated vision system that generates qualitative observations.

5. Implement a learning agent that uses the integrated representation and use it to assess the representation's strengths and weaknesses.

The initial goal of the research is to establish what types of action description and qualitative process representations exist. Some key questions to answer are: 'What are the limitations of the types of world behaviour they can express?', 'What are the trade-offs between learnability and expressibility?', and finally, 'How are the representational ideas assessed and evaluated?'. There is a rich history of research in both these domains on which to draw.

The second goal of the research is to identify the issues and environmental phenomena which make learning difficult. There are many features of realistic environments which make learning difficult. A number of these have been addressed in existing learning systems, however, no single learning system is able to handle them all.

The central goal of the research is to develop a new representation that can express both action effects and qualitative processes. This will build on the existing representations and re-use representational components as well as introducing new ones.

The aim of the representation is to allow an agent to construct generalised models of world behaviour that can be used to achieve goals in an environment which is observed in qualitative terms. It is desirable that the agent can do this in arbitrary and realistic environments. Because learning in realistic environments is difficult the representation should aim to find ways in which the burden of the learning system can be reduced.

The fourth goal of the research is to create a simulated environment which exhibits a rich and complex variety of behaviours. The simulation should allow an agent and a human teacher to interact with it in real-time, facilitating supervised learning. A skill system will provide the agent with a selection of high-level executable actions (e.g. pick up an object). The simulation should be able to implement the types of phenomena that were identified as making learning difficult.

A key component of the simulation will be the agent's vision system, which will provide a high-level qualitative description of the world in terms of objects and their properties and relationships. Determining the types of observations that the agent can perceive is non-trivial. For example, one difficult problem is that of describing the shape of an object in qualitative terms. This project will not attempt to resolve all these types of issues but the vision system should provide *sufficient* observation types such that interesting complex behaviour can be described non-trivially.

The final goal of the research is to implement a learning agent which uses the integrated representation. The agent should be able to learn generalised models from a few examples of observed action executions and qualitative object interactions. The agent should able to be given goals to achieve and use simple plans to achieve them (note that although the agent should be able to use the representation for planning, a detailed investigation into planning itself is not a goal of this project). The resulting performance of the agent will be used to assess the strengths and weaknesses of the new representation with regard to its applicability to learning in realistic environments.

1.2 Further Chapters

This section summarises the remaining chapters of the thesis.

Chapter 2: (Related Work) examines existing work in two representational frameworks: action description languages and qualitative process modelling. A variety of action representations are described in detail, including simple STRIPS actions, probabilistic actions, modular actions and others. These are followed by a description of qualitative process modelling and how it is used in qualitative simulation.

The chapter also describes a number of learning systems that have been built using the representations.

Chapter 3: (Q-Systems) describes the Q-System representation, a novel modular representation which integrates aspects of action models and qualitative processes. The ontological components of the representation are described, a simple notation is presented, and finally, the important design considerations are reviewed and discussed.

Chapter 4: (Simulation, Vision System & Skill System) describes the simulation that was developed in order to demonstrate Q-Systems. It describes a software architecture for the simulation, including the vision and skill systems which are used for real-time interaction. The simulation uses game engine software for calculating physical interactions and for rendering a 3-dimensional view of the simulated world.

Two different simulated worlds are presented: a toy world and a kitchen world. The chapter discusses the types of objects in the worlds, how they are observed and the types of interactions and behaviour that they can exhibit.

Chapter 5: (Histories) describes 'histories' - temporally and contextually restricted sequences of observation snapshots. Generation of histories is the first stage of a two stage learning process. An algorithm for generating histories from real-time observations is presented. The chapter describes various heuristics that can be used by the generator to control the types of histories that are output. The chapter concludes with an experiment in which various different combinations of heuristics are tested. The results are analyzed in order to find out which heuristics generate the types of histories that will be appropriate for generalisation.

Chapter 6: (Learning & Planning) describes generalising and planning algorithms that were used to test the Q-Systems representation. Three key components of the generaliser are described: a matching algorithm, a model refinement algorithm, and a model selection algorithm.

An agent that implements the learning and planning algorithms is also presented. The agent is run using the simulation, demonstrating its ability to learn generalised models and to plan with them to achieve simple tasks. The chapter concludes with a discussion of the limitations of the system.

Chapter 7: (Conclusion) summarises the contributions of the thesis with respect to the project's goals. It also draws together several insights regarding the issues of creating and learning the representation that were discovered during the project. The chapter finishes with a discussion of possible next steps in the development and evaluation of an integrated action and qualitative system learning.

Chapter 2

Related Work

2.1 Representing Actions

Actions are intentional, teleological changes to the world state. An agent that learns about the world must have an internal representation for describing when such actions are applicable, and the effects they will have on the world. This enables the agent to reach its goals by predicting how the world changes according to the actions it takes. Modelling actions is non-trivial, and this has resulted in the invention of a variety of symbolic action representations. The representations have varying levels of expressiveness, each one aimed at solving a different subset of the representational problems associated with action phenomena. Although each representation is different they share a common structure based on the simple notion of discrete state change, i.e. a symbolic action is a discrete event that causes the world to transition to a new state.

Action representations come in two main types: action logics and action description languages. Action logics use predicate logic to express both world states and world dynamics (where world dynamics are normally the effects of actions). Action description languages also use logic (or at least some restricted form) to express world states, however, action description languages differ in that world dynamics are expressed procedurally as state transition rules. This means that action description languages must specify their own rules for determining how the current state is updated to reflect the consequences of actions (and are interpreted according to the semantics of the action description language). Action logics on the other hand, use the rules of predicate calculus to specify and calculate the consequences of actions (which are interpreted according to the particular semantics of the action logic). Although the difference is subtle in terms of what can be expressed, there are advantages and disadvantages to both approaches. In particular, action logics have the advantage that consequences of actions can be calculated using logical inference (but consequently have the dis-advantage that actions must be carefully constructed to correctly express world dynamics). These issues are discussed in more detail in the following sections for relevant representations.

The remainder of this section begins with a description of two influential early papers that specify discrete state change type action representations. These are the Situation Calculus (an action logic) and STRIPS (an action description language). This is followed by descriptions of various extensions to the basic action representation and how they have been implemented. The strengths and weaknesses of each are assessed with respect to the task of learning in complex domains.

2.1.1 Actions as Discrete State Change

Actions in symbolic domains are modelled as discrete events that change the state of the world. An influential early attempt to model such actions is the Situation Calculus [26]. The Situation Calculus is an action logic in which an action is defined in terms of a pre-condition (a 'situation' in which the action may be carried out), and effect (the 'situation' after the action has been executed). The meaning of these attributes is intuitive: if the pre-conditions are true then the action may be executed; if the action is executed then the world will be in the new state specified. The following is a simple example describing the action of dropping an object. It states that the object can only be dropped if it is being carried and, if the object is fragile, it will break when dropped:

$$Poss(drop(o), s) \leftrightarrow is\_carrying(o, s)$$
$$Poss(drop(o), s) \wedge fragile(o) \rightarrow broken(o, do(drop(o), s))$$

Note how the situation variable 's' in conjunction with the 'do' predicate are used to define new situations inductively as a succession of action executions.

The Situation Calculus has difficulty specifying the non-effects of actions (i.e. the things that don't change), this is known as the 'frame problem'. The frame problem arises because a situation defines a precise and full state of the world according to the definition of the action that led to it. If a situation were to correctly capture the non-effects of an action, the action would require additional rules for every fluent that does not change. Intuitively we can avoid this problem by assuming that any fluents not specifically mentioned are unchanged, the so called 'common sense law of inertia'. There are a number of techniques that capture this intuition within the constraints of a formal logic (Default Logic [34] for example). Action description languages have the advantage of avoiding the frame problem. This is because new states are calculated according to the arbitrary rules of the language itself (rather than formal logic), and therefore the language can simply state that fluents that are not *explicitly* changed by an action remain unchanged in the subsequent state.

The STRIPS planning system [13] was published shortly after the Situation Calculus. The planner uses an action description language which has become known as the 'STRIPS' language. STRIPS actions, like the Situation Calculus, are represented as discrete state change, but rather than calculate action effects using the logic inference, they are instead calculated according to STRIPS own rules. STRIPS actions are represented as a precondition (a conjunction of assertions that must be true for the action to be executed), an add list (the assertions that are added to the current state when the action is executed) and a delete list (the assertions that are removed from the current state when the action is executed). The add and delete list together make up the effects of the action. States are defined as conjunctions of assertions. The following action schema defines a drop action, in which a held object 'X', that is above another object 'Y', will be on the lower object and no longer held, after it is had been dropped:

action:	DROP(X)
precondition:	HOLDING(X), ABOVE(X,Y)
add list:	ON(X,Y)
delete list:	HOLDING(X), ABOVE(X,Y)

STRIPS is computationally efficient in the sense that determining the consequences of an action is simply a case of adding and deleting from a list of assertions. However, the STRIPS representation has a number of limitations. Certain action phenomena cannot be expressed in STRIPS, for example there is no way to define an action with duration. Also, it makes the 'closed world assumption', meaning that there can be no unknown fluents in the world state. Despite such limitations, the STRIPS representation has been used as a template for the majority of action representations used in subsequently published planning systems.

Fig 2.1 shows a generalized view of a STRIPS like action as a state transition diagram. Action a is a discrete state change action with parameters p. State s is a state in which the precondition of the action is true. State s' is the new state subsequent to execution of the action with effects applied.

ADL[1][32, 33] is an extended form of STRIPS. It is designed to be more expressive than STRIPS for the purpose of representing more realistic planning problems. ADL actions are similar to STRIPS but allow for conditional effects, ambiguous effects and negative literals within effects. ADL

[1]Short for 'Action Description Language', from which action description languages get their name.

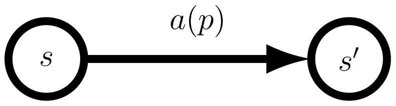


Figure 2.1: State transition diagram for a simple STRIPS like action.

also removes the closed world assumption (i.e. there may be unknown literals in states). The following is an example of an ADL action:

> $Action(Drop(x : Object)$
> Precondition: $holding(x)$
> Effect: $\neg holding(x)$
> **when** $fragile(x) : broken(x) \vee cracked(x))$

This version of the drop action states that an object must be held before it can be dropped and that the object will not be held after it is dropped. It also defines an effect that is both conditional and ambiguous: if the object is fragile then it will be broken or at least cracked when dropped. This particular effect could not be expressed in STRIPS.

Fig2.2 shows transition diagrams for the additional action phenomena that can be expressed using ADL. Conditional effects can be viewed as a set of transitions where each state pair $s_i \rightarrow s'_i$, represents a different conditional outcome. Ambiguous effects are shown as two or more non-deterministic transitions.

A further extension to the STRIPS family, the 'Planning Domain Definition Language' or PDDL [27], was developed to allow planning researchers to exchange benchmark problems using a standard notation. PDDL supports both STRIPS and ADL action operators. Later revisions of PDDL introduce a number of extended action effects that are discussed in the following sections.

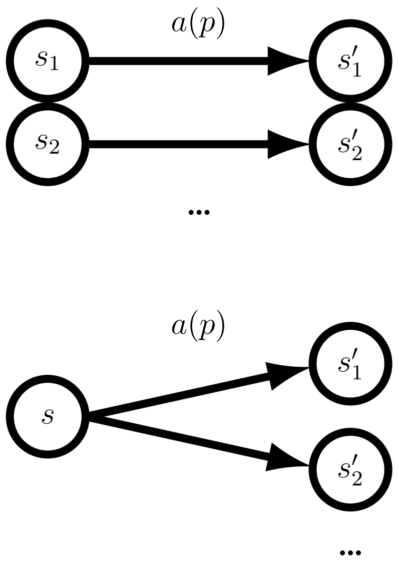


Figure 2.2: State transition diagram illustrating conditional (top) and ambiguous action effects.

2.1.2 Hierarchies of Actions

Actions can be used to represent arbitrarily complex activities. For example the action *fly plane to destination x* is vastly more complex than the action *press cabin light switch*. Action representations do not generally distinguish between complex and simple actions. However, it is sometimes useful to define complex actions as a composite of simpler actions. Action composition naturally leads to hierarchies of actions such as those used in hierarchical task networks (HTNs) [11].

The action language used in PDDL supports HTN like action hierarchies. The effect of executing a PDDL hierarchical action is defined as the combined effect of executing each sub-action, either in series or parallel. For example, the complex fly plane action can be decomposed into simpler *take off*, *cruise*, and *land* actions (each of which could also be further decomposed).

Fig 2.3 illustrates a hierarchy of composite actions as a state transition diagram. The composite action $comp_1$ is a sequential composition of actions a_1 and a_2. Composite action $comp_2$ is a composition of actions including $comp_1$ and a_n. The hierarchy has n simple actions when fully decomposed.

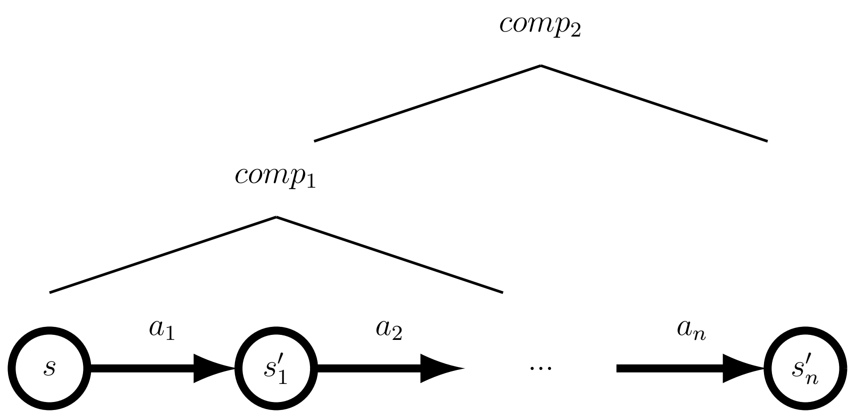


Figure 2.3: State transition diagram showing a hierarchy of sequential composite actions.

Note that it is only the lowest level 'atomic' actions in a hierarchy that specify effects, the effects of composite actions must be inferred from the effects of their children. It is generally the case in most planning domains that atomic actions cannot be interrupted, this is not true of composite actions where special consideration must be given to pre-conditions being invalidated during execution.

2.1.3 Actions with Duration

Actions are not instantaneous, even simple actions such as pressing a switch have some duration. The action representations considered so far, do not account for actions taking different lengths of time to execute. This is reasonable so long as the intermediate effects of a long action are not important (i.e. only the start and end state of the action are important), and also if there is no requirement to compare the lengths of different actions. However there are many problem domains where some form of representation of duration is required in order to usefully model actions.

The 'action' of pouring liquid from one container to another can be considered a durative action in which the effects during execution are important. For example, we may wish to know the amount of liquid remaining at a given point during the action, this information could be used calculate when the action will end. One way of modelling this type of action is to represent the durative action in terms of non-durative start and stop actions. E.g. the durative action *pour liquid* becomes *start pouring* and *stop pouring*. The start and stop actions are short enough to be considered instantaneous. The start action adds the intermediate effects of the action and the stop action removes them. In the previous example, *start pouring* would add the effect that liquid is currently flowing from the first container. Modelling durative actions in this manner avoids the need for any additional representation but with the drawback a that 'durative action' must be inferred from the combination of two related start and stop actions. Fig 2.4 illustrates a durative action modelled as complimentary start and stop actions. Action a_{start} starts the action and adds appropriate effects, a_{end} finishes the action some time later. Other actions/transitions may occur between the start and end actions (in fact there is no constraint that the end action must ever occur). Since no additional representation is used, any action representation may adopt this convention to model durative actions.

An alternative way of representing durative actions is to use an extended action schema. Extensions allow specification of a durative action in terms of: its start and stop actions; pre-conditions that must apply at different points during the action; effects that are applied at different points

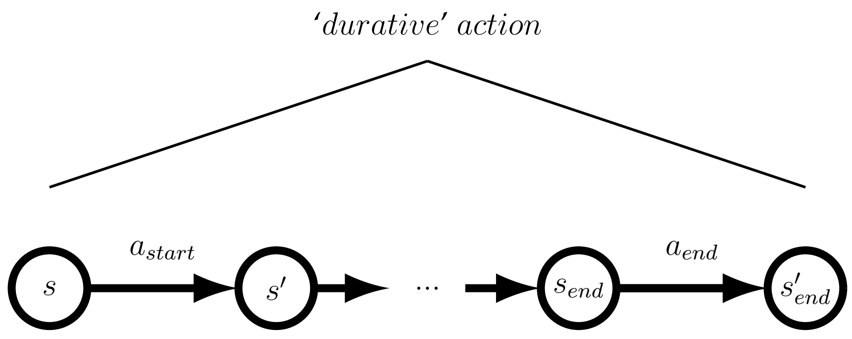


Figure 2.4: A durative action modelled as two 'instantaneous' start and stop actions.

during the action; and finally, the amount of time its takes to execute the action. This type of schema is used in the PDDL 2.1 [17] planning language, it allows for conditions and effects to specified at the beginning and end of the action. Intermediate effects are calculated using the start state and time elapsed. A durative *fly* action can be expressed in PDDL using the following schema:

```
(:durative-action fly
   :parameters (?p - airplane ?a ?b - airport)
   :duration (= ?duration (flight-time ?a ?b))
   :condition (and (at start (at ?p ?a))
      (over all (inflight ?p))
      (over all (>= (fuel-level ?p) 0)))
   :effect (and (at start (not (at ?p ?a)))
      (at start (inflight ?p))
      (at end (not (inflight ?p)))
      (at end (at ?p ?b))
      (decrease (fuel-level ?p)
         (* #t (fuel-consumption-rate ?p)))))
```

The schema specifies the duration of the action as the flight time of plane $?p$ travelling from location $?a$ to $?b$. The 'over all' operator is used to specify invariant conditions that must be true during the action, e.g. the plane has not run out of fuel. Unlike simple actions, effects are specified for *both* the start and end of the action. The intermediate effects of the action are specified in the bottom line which states that the amount of fuel on the plane, at any given point during the action, is equal to the time elapsed since the action started, multiplied by the fuel consumption rate of the aircraft.

Specifying actions in this way can lead to complicated interactions between actions. At any given point in time a number of different actions may be executing concurrently and all their intermediate effects must be combined to give the current world state. In the aircraft domain we could imagine a number of different actions that affect the fuel level either directly (such as mid-flight refuelling) or indirectly via the fuel consumption rate (such as increasing speed). Careful consideration must be given to concurrent effects when representing actions in this way.

Fig 2.5 illustrates a PDDL 2.1 style durative action. s' is the state with start effects applied. The dotted transition represents an arbitrary number of intermediate states with ongoing effects applied. s'' is the final state with the specified end effects applied.

2.1.4 Concurrent Actions

Many problem domains assume that actions do not occur concurrently, this is an unrealistic assumption for real world applications, especially in environments where multiple agents are operating (e.g. robots in a factory). A fully expressive action description language must specify how the world changes if two actions are being executed concurrently. This is further complicated when durative actions are allowed because they may specify different effects are different times during their execution.

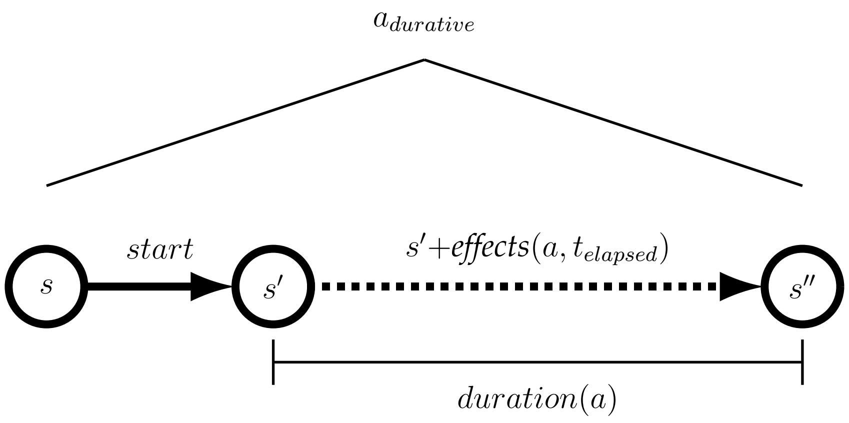


Figure 2.5: A durative action modelled with a fixed duration and intermediate effects.

A simple solution to concurrency is to specify that the world state is undefined when actions occur simultaneously, obviously the quality of this solution degrades as the frequency of concurrent actions increases. Another simple solution is to specify that the effects of both actions be applied in some arbitrary order. This works in many cases but fails when the effects of actions interact (i.e. the effects of one action can 'undo' the effects of the other).

PDDL 2.1 allows its durative actions to execute concurrently. Start and end effects are prevented from occurring concurrently, however the intermediate effects are allowed to interact. For example, the effect of concurrently applying two heat sources as part of two different boil water actions causes the duration of the actions to be reduced.

An alternative approach used in [24] is to allow for all possible combinations of effects resulting from concurrent actions to remain valid. Predicting the result of concurrent execution leads to not just one state but a set of possible states. The uncertainty can be resolved when the actions are actually executed.

2.1.5 Delayed and Hidden Effects of Actions

The effects of some actions are not immediately observable (if at all) to an agent. For example, the effect of turning off an outside light using an inside switch, may not be directly observable by the agent at the time of executing the action. The agent can of course subsequently observe the change by moving outside. Action representations do not explicitly model the fact that an action effect may be hidden to the agent. This is because either problem domains are restricted to being fully observable (or at least the subset of the world affected by the current action is fully observable), or it is assumed that *any* effect may be hidden and therefore appropriate mechanisms are in place to deal with this, outside of the action representation.

'Delayed' effects are closely related to hidden effects. A delayed effect is an observable effect that reliably occurs sometime after an action is executed, and is a direct consequence of some hidden effect of the action. For example, the act of poisoning some weeds may have no directly observable consequences, however the weed dying after some delay is reliably observable due to the hidden effects of the action. Delayed effects need not be represented explicitly since they are an indirect consequence of an action and can be inferred from knowing the direct effects of the action (i.e. we can determine the fact that the weeds will die from the fact that they are poisoned regardless of how the poison got there). It could be argued that explicit representation of delayed effects would be useful however there do not appear to be any examples of this in existing action description languages.

2.1.6 Probabilities in Actions

Many actions have uncertain outcomes. For example, the act of throwing a paper ball into a bin may or may not result in the paper ball ending up in the bin. Given enough information the outcome could be calculated to a high degree of certainty (information such as the precise speed and movement of the throwing arm, the current air speed, etc), however this kind of calculation will always be impractical for some applications. Action representations can model uncertainty by giving probabilities to different outcomes. Oates' MSDD learning system [31] uses a type of probabilistic action outcome in a propositional domain. The outcome of each action is given a probability of success depending on the state in which in the action is executed. The action can either succeed or fail, so the probability of failure for a given pre-condition is $1 - p(success)$. The following schema is an example of a probabilistic action using MSDD's representation:

<pickup, gripper_dry, not_hold_block, hold_block, 0.98>
<pickup, gripper_wet, not_hold_block, hold_block, 0.49>

This action states that a block will be picked up successfully 98% of the time *if* the gripper is dry. Conversely, if the gripper is wet then the block will be successfully picked up only 49% of the time.

A limitation of MSDD's action representation is that it only distinguishes between success and failure of an action, rather than a range of possible outcomes. This is addressed by the representation used in Zettlemoyer et al's more recent learning system which is described in [45]. Their representation allows for an arbitrary number of different outcomes for a given action and pre-condition. Further, they allow for a special 'noise' outcome which is used to represent the possibility of unusual outcomes in which the world state is difficult to model. In these cases the subsequent world state becomes unknown; noise outcomes are considered rare enough to allow for the occasional unpredictable outcome. The following schema is an example rule from [45]:

$pickup(X) : \{Y : on(X, Y), Z : table(Z)\}$
$inhand\text{-}nil$

$$\rightarrow \begin{cases} .80 : \neg on(X,Y), inhand(X), \neg inhand\text{-}nil, clear(Y) \\ .10 : \neg on(X,Y), on(Z,X), clear(Y) \\ .05 : \text{no change} \\ .05 : \text{noise} \end{cases}$$

This pickup action has four possible outcomes: an 80% chance of correctly picking up the block; a 10% chance of dropping the block on the table; a 5% chance of the block remaining in its current position; and finally, a 5% chance of something unusual happening (perhaps the gripper arm knocks several blocks onto the floor).

Figure 2.6 illustrates probabilistic actions, of the type just described, using a state transition diagram. An arbitrary number of transitions are each given a probability which sum to 1. The state marked '?' represents a noise outcome. A 'no change' outcome is shown as a self loop with a specified probability.

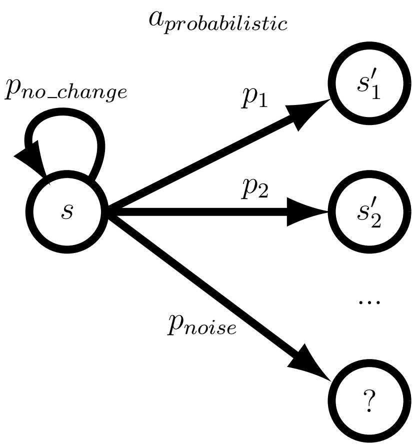


Figure 2.6: State transition diagram illustrating probabilistic action outcomes.

2.1.7 Modular Actions

Many actions share common features, for example, walking, climbing and driving a car all involve moving from one place to another. These actions could all be thought of as special cases of a more general 'move' action. Modular action representations attempt to exploit this commonality by modelling collections of reusable general actions which are 'inherited'[2] by specialized actions.

General or 'parent'[3] actions describe preconditions and effects as normal, these will be automatically included by any inheriting 'child' actions. The child actions specify further preconditions and effects, and these may reference objects that are bound to objects in the parent action. Bruce et al [7] were first to apply these types of actions to a STRIPS planning domain. The following schema are examples of specifying inheritance using their representation:

```
put(Obj,Container):  isa  moveThruPortal(Obj,Portal)
                     where  portal(Container,Portal)

get(Obj,Container):  isa  moveThruPortal(Obj,Portal)
                     where  portal(Container,Portal)
```

The two actions get and put model the actions of putting an object Obj into a container and removing it. By themselves, they do not model the situation in which the container has a lid which can be opened or closed. This could be remedied by creating new actions with additional preconditions and effects, alternatively the actions can inherit from a $moveThruPortal$ parent action (as is the case in the example above). $moveThruPortal$ specifies the additional precondition that the opening of $Portal$ (bound to $Container$) must be open. Thus creation of entirely new actions is avoided.

[2]The term 'inheritance' is used in this context in much the same way as it would be in the world of software engineering (from which modular representations take inspiration).

[3]The term 'component' is also used.

A similar style of modularisation is used in the Modular Action Description language [25], in which modularity is added to the $C+$ causal action language. Although the action descriptions are somewhat different to STRIPS, the modularity aspect is applied in much the same as way as the example above.

In general, modular representations allow large action sets to be represented efficiently, this is especially true when attempting to represent very general actions (such as 'put') where the number of different preconditions and effects is virtually unbounded. Another advantage is that modularity allows knowledge from one problem domain to be easily transferred to another. The author of the Modular Action Description language describes it succinctly as "[a mechanism for] factoring out the common elements of specific action domains".

2.2 Learning Actions

This section describes published work regarding systems that learn the effects of symbolic actions. An action learning algorithm addresses the problem of extracting a set of action schema from a set of observations, which are usually in the form of example action executions. A particular algorithm is shaped by a combination of three factors: the action representation used, the complexity of the environment from which examples are drawn, and the types of observations provided. These constraints together determine the difficulty of the learning of the learning task. An action learning task can be made trivial in a highly constrained environment, equally, the task can be made very difficult in a relatively unconstrained environment. Creating an agent that can learn expressive actions, in unconstrained environments is the overall goal of the research presented in this section. However, given the difficulty of this task, current algorithms typically allow only a few constraints to be dropped, for example, hidden effects may be allowed but not probabilistic effects.

Learning symbolic action effects in complex environments is non-propositional or 'relational' in nature. This is in contrast to the majority of machine learning research in which patterns must be discovered from propositional examples described in terms of a fixed set of features. Relational learning requires extracting patterns where each example involves an arbitrary of number of 'entities', which are described using their individual attributes and also their relationships with other entities; relational data is typically expressed in first-order logic. When applied to learning action models, the entities are the objects in the environment on which an agent can act, or which can be affected by actions. Learning relational models in unconstrained environments is an unsolved problem and active area of research in artificial intelligence. The algorithms presented in this section derive from generic approaches to relational learning but are adapted to the specific problem of learning action models.

An important distinction in action learning systems is made between those that learn autonomously and those that are guided. Guided learners may rely on instructions or demonstration from a teacher, while others use expert planning information. Obviously this extra input is advantageous to learning but may not always be available depending on the particular problem domain. On the other hand fully autonomous learners rely purely on their own actions and observations.

Examples of action learners include OBSERVER [40] which adapts version space learning [28] to the task of learning the preconditions of actions. The SLAF algorithm [2] also uses a form of version space learning to learn about actions with hidden effects. In this case the version space becomes the set of possible action-state transition sequences. A statistical relational learning approach is taken by Zettlemoyer et al [45] to learn probabilistic action outcomes. The algorithm combines inductive logic programming [30] concepts with Bayesian statistical learning. A form of reinforcement learning [41] is used by Thomaz [38] to 'teach' an agent when to apply certain actions. The remainder of this section describes these action learning systems and others in greater detail.

2.2.1 Probabilistic outcomes in a simulated 3D world

Zettlemoyer et al. [45] describe a method for learning extended STRIPS like rules in a noisy stochastic world. An agent is situated in a simulated 3D world with realistic physics. The world contains a number of blocks of various sizes and the agent must learn the effects of manipulating them with its gripper arm. The task is difficult because there are multiple possible outcomes for a given action, e.g. blocks may fall over when stacked or the gripper may accidentally drop a block. However the learning task is simplified by making certain assumptions: the world is completely observable; the world can only be changed by the agent's own actions; actions are atomic, i.e. each action has only a small number of discrete effects that are observed simultaneously and immediately after the action.

A probabilistic action representation is used to represent the effects of actions. The extended rules incorporate deictic references, probabilistic effects and noise outcomes (see section 2.1.6 for further detail). The actions are used to model the inherit uncertainty of interactions in the simulated world. Action preconditions are described in first-order logic, with the addition of a counting quantifier and a transitive closure operator. These additions are used for defining derived predicates. For example, the concept of an object being 'above' another can be derived from the transitive closure of 'on' and is written:

$$above(X, Y) := on * (X, Y)$$

Similarly, a 'height' predicate is derived by counting the number of blocks below the block of interest. This is written as follows using the counting quantifier '#':

$$height(X) := \#Y.above(X, Y)$$

Derived predicates such as these give the rule language greater expressiveness and allow it to represent useful collections of relations more concisely.

The system discovers action models using a learning algorithm which attempts to find the best set of rules given a set of training examples. Each example contains an action and the (fully observable) world states directly before and after the action (written as a triple (s, a, s') where s' is the state directly after the action a is applied in state s). The algorithm starts with a single default rule that predicts only noise, and performs a greedy search through the space of possible rule sets. A rule set is heuristically scored according to a metric based on a MAP (Maximum A Posteriori) estimation of the probability of the rule set given the examples (the rule-set's prior is determined by the complexity of the rule-set, therefore favouring concise rule sets). The score of an individual example (s, a, s') against a given rule set is calculated as the log of the probability specified for s' in r, where r is the rule that matches s and a.

A key component of the algorithm is the group of search operators that create new candidate rule sets by inventing, dropping or modifying existing rules. New rules are created by taking unexplained examples from the training set and creating specific rules to cover them. Rules are generalized or specialized by arbitrarily dropping or adding components to rule preconditions and inferring new outcomes. Finally, entire rules can be arbitrarily dropped during the search.

A significant limitation of the learning procedure is that it relies on the fact there are a limited number of objects. For example, one method for creating new rule candidates is to add references to arbitrary objects, which makes sense in a world with a small number of objects but not in a realistic world where there are an unbounded number of objects to choose from.

The quality of the learned rules was assessed by applying them to a problem task (building towers in this case), rather than comparing them to some idealised hand crafted model as is done for some other algorithms.

Notwithstanding the restricted problem domain, the agent successfully learns rules that perform as well as hand constructed rules provided by an 'expert' (a human adult). The learned rule set was used by a planner to build high stacks of blocks, and it achieved stacks of over 15 blocks tall. The following is an example learned rule:

$$\begin{array}{l} \mathit{putOn}(X) : \{Y : \mathit{topstack}(Y,X), Z : \mathit{inhand}(Z), T : \mathit{table}(T)\} \\ \mathit{size}(Y) < 2 \\ \qquad\qquad \rightarrow \begin{cases} .62 : \mathit{on}(Z,Y) \\ .12 : \mathit{on}(Z,T) \\ .04 : \mathit{on}(Z,T), \mathit{on}(Y,T), \neg \mathit{on}(Y,X) \\ .22 : \text{noise} \end{cases} \end{array}$$

The example shows that the agent has learned that placing blocks on stacks in which the top block is small (< 2), is not a very reliable action as approximately 1 in 3 attempts are unsuccessful.

An interesting result is that the algorithm is much more effective when noise outcomes are allowed (the outcomes which specify a rare and difficult to model action result). The learner avoids over-fitting when it is not required to explain every possible outcome. Another interesting result is that the use of derived predicates (e.g. *above* and *height*) was essential for generating successful rules.

Overall, the agent can learn good quality rules in a world which simulates some specific complexities found in a realistic world (noisy, probabilistic outcomes) but relies on a number of simplifying assumptions with regards to the environment, it does not have to deal with large numbers of objects, other agents, hidden effects, or delayed effects.

2.2.2 SLAF: learning in a partially observable world

The 'SLAF' (Simultaneous Learning And Filtering) algorithm [2] addresses the problem of learning in a partially observable world. The goal of the system is to learn the deterministic direct effects of actions even when they are not immediately observable. A typical scenario involves a number of light switches in a room that control light bulbs in *other* rooms; the agent cannot determine the effect of using a switch without moving to the other rooms and observing the actual state of the associated light bulb.

The SLAF algorithm resembles version space learning. The algorithm maintains a set of valid action descriptions that are consistent with the set of possible transition belief states (where a transition belief state is a sequence of world states with an action associated to each transition). The algorithm works in an on-line manner by adding to or deleting from the set of belief states for each new observation. For example, consider the following sequence: the agent flips the switch, the agent moves to the lit room, the agent observes the bulb is on. The agent can now safely eliminate the belief state in which the 'flip' and 'move' actions result in the bulb being off. Finding an accurate belief state is simple for small state spaces but soon becomes intractable. To overcome this Amir introduces a more compact rule based representation for belief states that allows for efficient evidence based update. The rules used are similar to STRIPS rules but allow hidden effects to be included as part of the description.

The algorithm successfully learned a number of rules in a partially observable domain. It demonstrated that on-line incremental refinement of action models, as evidence is encountered, is a valid approach to learning in non-trivial domains. However, the system is limited to solving the single issue of partial observability, it fails to address issues such as probabilistic action effects. The algorithm is also particularly dependent on the assumption that there are no other agents that can make changes in the world. Consider an example in which a second agent turns off a light in another room. The first agent will maintain the transition belief state that includes a sequence of events in which its actions cause the light to be off. This invalid transition belief state will in turn result in the learned rules allowing for the incorrect effect of the light being off. Clearly this problem gets much worse as the number of agents and their level of activity increases.

It is not obvious how the effects of such exogenous actions would be excluded from the transition belief states. One possibility is to include additional frequency data in the transition belief states. This would allow culling of sequences that appear infrequently (i.e. the ones in which other agents actions are causing confusion). The drawback of this approach is that learning would require a greater amount of experimentation (i.e. the agent would have to perform the *switch light on → move to room → observe light* sequence several times before it is used as evidence of a hidden effect).

2.2.3 MSDD: probabilistic outcomes with exogenous events

Oates and Cohen [31] present an agent capable of learning in a world with probabilistic effects, context-dependent effects, and exogenous events (i.e. events caused by other agents). The learning scenario involves a robot with two arms painting blocks. The robot has a fixed number of input sensors (e.g. the agent can sense that block 'X' is painted) and actions (e.g. paint block 'X'). The learning task is to create STRIPS-like operators for each action where an operator describes the probable outcomes given the current state of the world. Each operator assigns an estimated probability to each possible outcome (see section 2.1.6 for details of the action representation).

The 'MSDD' algorithm finds operators by performing a best first search through the space of all possible contexts and effects starting with a most general operator (i.e. the operator specifies that: 'any sensor can take any value', given the precondition that: 'any sensor has any value'). An observation trace of actions and world state transitions is used to evaluate operators. The operators are heuristically scored by counting how often the context and effect occur together, plus a bonus for a frequently occurring context. The search space is pruned by removing illegal and duplicate operators. High scoring operators are output along with a probability of success (the probability is simply calculated from the number of successes in the observation trace). New search nodes are generated by adding new operators that are specializations of existing operators.

The agent was able to learn a full set of operators for the simple blocks painting world using the algorithm described. It was able to do this even when a number of exogenous actions were included in the observation trace. Exogenous actions are modelled as 'noise' streams in which random observations are added to the world state. These noise streams are not a realistic model of exogenous actions for two reasons: first, actions by other agents are not simply random (note that the search algorithm uses frequency of pre/post state correlations to guide the search); and secondly, actions by other agents should be able to interact with the same objects as the learning agent (this too will have an impact on the correlation heuristic). Both of these limitations make the learning domain a less realistic model of real world complex environments, and certainly simplifies the learning task.

The agent's fixed number of actions and sensors was very limited (less than 10 of each) and can be considered a classic 'micro-world'. The authors suggest that the search time grows linearly as the number of noise streams increases. The claim is supported empirically for upto 20 noise streams. There is no additional evidence to suggest that this will be true if the number of *sensors* is increased significantly into the thousands (a real world environment can easily generate this number of observables).

Other limitations of MSDD (as applied to learning actions) are the assumptions that effects occur immediately after an action and that the world is completely observable; this precludes the agent from learning about actions with delayed or hidden effects.

2.2.4 EXPO: detecting and refining incomplete actions

Gil [18] describes a proactive experimental approach to learning action operators. The problem addressed is that of refining incomplete STRIPS rules that have missing preconditions or effects, for example, an 'open door' action is missing a precondition such that the door must be unlocked for successful execution. The system, named 'EXPO', works in an online manner by selectively and continuously monitoring the world during plan execution (plans are created and executed within the PRODIGY [39] agent architecture). The agent is monitoring for either of two kinds of failures that indicate that an action rule is incorrect. One such event is observing an unexpected outcome immediately after an action execution, e.g. an *open* action was executed and the door didn't open as expected. In this case EXPO attempts to find a new precondition for the action. The precondition is found by drawing up a list of candidates (derived from successful past executions) and experimenting with each to discover the correct one.

The second type of failure is observed when a precondition for the current action (as predicted in the plan) is false. In this case EXPO determines that a previously executed action has a missing effect involving the predicate specified in the required precondition. The action with the missing effect is found by experimenting with all the actions executed since the predicate was last observed. The incorrect action is then refined by adding the predicate as an effect. In this manner EXPO incrementally refines its action rules.

This technique was successfully applied to a small deterministic problem domain with many incorrect rules. One advantage of EXPO is that it only refines rules for situations that are actually encountered (because learning is triggered during plan execution). This allows EXPO to avoid having to create action schema for every situation in which a rule *might* be used. A general purpose action, such as *put on*, has a potentially unbounded number of rare and esoteric situations for which pre-conditions and effects must be specified in order to learn a 'perfect' model. An unguided learner may learn a number of rules that are correct but never actually used.

The concept of incremental refinement through experimentation that is guided by plan execution failure has some advantages over off-line and unguided approaches to learning in complex environments. Unfortunately, the EXPO system makes a number of assumptions that limit its particular implementation to micro worlds. For example, introducing exogenous events would confuse EXPO (and cause it to learn incorrect rules) because it expects only its actions to affect the world. Another weakness is that EXPO requires approximately correct rules to begin with, it cannot learn from zero knowledge of actions, unlike some other action learning systems.

2.2.5 OBSERVER: learning from plans and experiments

Wang [40] describes an action learning system called 'OBSERVER'. It uses a combination of domain expert traces (a sequence of actions with pre and post states performed by a human with knowledge of the problem domain) and simple experimentation to learn STRIPS-like rules. The task for OBSERVER is to learn new operators along with their preconditions, effects and conditional effects. The scenario presented is that of a manufacturing plant containing various tools, objects and actions for manipulating them.

OBSERVER learns the action preconditions and action effects separately. Preconditions are learned by incrementally refining a most general and a most specific set of preconditions for the given operator (similar to version space learning). The final precondition is found when the two sets converge. Effects are learned by incrementally generalising the delta states (the post-state minus the pre-state) of a given action. Conditional effects are learned when literals in a delta state cannot be unified with those in the effects (the algorithm assumes that all changes in any delta state must be included in the action effects).

An individual pre-condition from the most specific pre-condition is generalized if it can be matched to exactly one assertion from an observation pre-state. Matching is restricted to unification of only a single constant from each pre-condition. This is done to reduce the complexity of the finding potential matches in large states (with the penalty that some potential matches may be missed). Pre-conditions are removed from the most specific pre-condition if they cannot be matched to any assertions in the pre-state.

The most general pre-condition is specialized by adding 'necessary' and 'critical' pre-conditions. Necessary pre-conditions are generated from the most specific pre-condition, whereas critical pre-conditions are generated from example pre-states. Necessary conditions are found when an action fails and only a single (necessary) assertion from the most specific pre-condition is *not* met in the pre-state. Critical pre-conditions are found by comparing the pre-states of an action failure and success. If there is only a single assertion from the most specific pre-condition, which is true when the action succeeds, yet false when the action fails, then it is considered critical.

OBSERVER demonstrated that it can learn rules that are equally as useful as a set of hand coded expert rules. Interestingly, the expert observations alone were not sufficient to learn good rules, it was only after refinement by simple experimentation that the rules were sufficiently improved. Despite this success, the problem domain for observer is one of the more limited described so far. For example, it lacks probabilistic effects, noisy observations, delayed effects and other agents actions.

2.2.6 LIVE: integrating learning and problem solving

Shen's learning system [36], unlike most other action model learners, integrates problem solving and learning into a single process. The agent uses STRIPS-like rules to create plans that will achieve its goals. When the plans fail or a plan cannot be constructed, the agent will refine its model. The model is refined in an incremental manner by creating new rules or adjusting preconditions and effects. The algorithm has been applied to a number of limited abstract problems, the famous Tower of Hanoi puzzle for example.

The agent employs a series of different stages to learn its rules. The first is rule creation, followed by rule refinement and lastly term construction. Initial approximate rules are constructed by exploring. Exploration is directed by choosing actions that either affect features used in the agent's current goal, appear to have no effect, or are simply randomly chosen from a pre-defined list. New rules are naively constructed by observing the effects of these exploratory actions. The new rules are then used in the agent's plans.

If an action execution fails then the associated rule is revised. This is done by first creating a 'sibling' rule based on the failing rule, but with a different precondition and the unexpected effects. The sibling's precondition is found by searching a history of rule executions for differences between the pre-state of previous successful executions and the current world state. The two rules are then revised together. Revisions are made by generalising the condition of the new specific rule sibling and specialising the condition of the old rule.

The final stage of learning is that of term construction. This allows the agent to create derived predicates using the agent's 'innate mental relations' such as '=' and '>'. For example, a non-directly observable relation 'SIZE>' can be derived from the directly observable 'SIZE' predicate and '>' operator. Furthermore, the agent has some ability to create terms where there is no observable discriminating predicate. For example, in the Tower of Hanoi domain, LIVE can create a pre-condition that corresponds to 'the most recent disk put on post X'. This is achieved by detecting sequences of actions that imply the pre-condition is satisfied. For example, the action $pickup(X, Post)$ can have the pre-condition: 'disk X was the most recent disk put on the Post', because it has access to the history of 'put on' actions that led to the current state.

LIVE's method of learning is robust in some ways and brittle in others. Its robustness arises from the ability to learn autonomously while achieving goals, its ability to invent new terms for use in preconditions, and the fact that rules are gradually refined over time. Its brittleness comes from a number of simplifying assumptions about the problem domain: the effects of actions must be deterministic; the number of objects and observable features is limited; there can be no noise in the observations. However, the approach of combining exploration, experimentation and incremental model refinement is quite powerful since it gives the agent a level of autonomy that other action learners do not have. The agent also does not require outside direction (from a teacher for example) when deciding which actions to employ in order to improve its rules.

2.2.7 ARMS: learning from plans with incomplete observations

All of the agents previously described rely on observations of the world before (pre-state) and after (post-state) action execution (where the world is either partially or fully observable). The ARMS algorithm [44] takes a different approach by learning an action model from plan examples where only the pre/post states are known for the plan *as a whole*. No intermediate observations are required. The motivation for ARMS comes from a requirement to learn from experts where actions are easily recorded but world states are not.

The algorithm generates a complete set of action pre-conditions and effects. It learns from a set of example plans (action sequences) along with initial and goal states. The algorithm works by casting the problem as a weighted propositional satisfiability (SAT) problem. The SAT is constructed using constraints to represent preconditions and effects, the fixed structure of action models, and constraints imposed by the plan. Some example constraints are: 'all actions must have at least one effect', 'clear(x) is a precondition of lift(x)', 'the final action in a plan must have one of the goal predicates as an effect'. Fully solving the SAT is computationally hard so constraints are weighted to prioritise actions and predicates that occur frequently together in the plan examples. The algorithm proceeds by generating a subset of likely actions by partially solving the weighted SAT. The algorithm then updates the plan examples by executing the actions discovered so far. The process is repeated with the now shorter plan examples until all action rules are found.

ARMS is able to learn basic STRIPS rules (i.e. no conditional effects, no probabilistic effects, etc) from sets of example plans, however the learned action models are imperfect and include some incorrect preconditions and effects. ARMS is also limited to off-line operation because it requires a substantial number of example plans to learn from. It is not clear how ARMS could be used in an on-line scenario where action models must be refined incrementally as new evidence is encountered. The ARMS approach does however raise the interesting question of what other types of evidence can be used to construct action models, when complete action sequences with pre and post states are not available.

2.3 Representing Qualitative Behaviour

This section describes qualitative representation, an area of artificial intelligence that is concerned with reasoning about complex systems without using quantitative data or formulae. Qualitative models are specified in terms of qualitative variables. The range of a qualitative variable is a division of the space of real numbers into contiguous ranges. All of the real values falling within each range are treated together as a single qualitative value. This is in contrast to a quantitative model in which every possible real number is treated as distinct. The number of possible values for a given variable is finite and usually quite small. Although this quantization loses information, the precision can be sacrificed without losing the ability to model interesting behaviour. Obviously, what is 'interesting' depends on the particular problem domain, and there are of course many problems for which qualitative modelling is not sufficient.

Qualitative models allow variables to have qualitative relationships or constraints. These describe how a change in one qualitative variable will affect another, but only in qualitative terms. For example, it can be stated that one qualitative variable increases monotonically with another; however, the precise (quantitative) increase cannot be calculated from the qualitative relationship. A group of interacting qualitative variables and constraints can be used to specify a qualitative process. The behaviour of a process can be simulated by starting with an initial state and iteratively growing a graph of qualitative state transitions. The graph is grown by adding all successor states that are consistent with the specified constraints (for example, a system cannot transition to a state in which two inversely proportional variables are both increasing).

A qualitative simulation can be used to predict the possible state transitions that can occur within a system and under which circumstances. If the qualitative ranges for variables are well chosen then it is possible to simulate complex systems, and discover interesting behaviour, without resorting to quantitative methods.

There are two primary motives for choosing qualitative over more precise quantitative modelling. The first motive is that for some problem domains, quantitative analysis is either impossible or too difficult. One reason for this may be that sufficiently precise measurements cannot be obtained for the system under analysis. For example, in one domain it may not be possible to measure the speed of an object exactly enough to calculate the resulting behaviour. In this case, simpler qualitative measurements, such as the speed being greater than zero, can be used to predict more abstract behaviour. Similarly it may be the case that precise quantitative relationships are not known. For example, the precise quantitative formula describing the effect of air temperature on the speed of a plane may be unknown. In this case a qualitative relationship may be used in its place. Finally, even if accurate quantitative measurements and formulae are available, it may be too computationally expensive to simulate the system's behaviour accurately. In this case a more computationally efficient qualitative simulation may be the only option.

The second motive for choosing qualitative modelling is that quantitative precision may not be required. Qualitative results may be sufficient for the problem domain in question. One reason for this is that only qualitative results are of interest. For example, a pipeline designer may wish to be assured that flow can only ever occur in one direction between two points and never the other way around. A qualitative simulation could answer this question even though it could not calculate the exact quantitative amount of contra-flow (if any exists). Similarly, it may be the case qualitative methods are sufficient for spotting anomalies which are then analysed quantitatively in isolation (this may be more efficient than performing a full quantitative analysis on an entire system). Finally, qualitative methods may be considered 'good enough' for common sense reasoning in everyday environments. People perform many complex tasks and analysis without resorting to complex quantitative analysis. The process of cooking some spaghetti can be achieved without calculating the precise heat flow from the stove to the water. A full quantitative analysis may make the process a little faster (knowing exactly when the water will boil, using precisely enough water for the spaghetti to cook) but is not necessary in this problem domain and others like it.

Qualitative Process Theory [14] was an early formalisation of qualitative modelling concepts, it brought together three inter-related streams of qualitative research: common sense reasoning, as described in the naive physics manifesto [20], representation of quantity in symbolic reasoning, and finally, simulating qualitative behaviour through 'envisionment' [10]. The theory has subsequently been used to model a multitude of different systems and draw useful conclusions about their behaviour. The remainder of this section describes the main techniques used to represent qualitative systems.

2.3.1 Quantities, Constraints and Processes

Qualitative models are represented in terms of qualitative variables, constraints and processes. A qualitative variable is specified using a quantity space. A quantity space is a 'finite, totally ordered set of symbolic landmark values representing qualitatively important values in the real number line. The value of a qualitative variable at any given point in time, is either a landmark value or a region between landmark values. A simple quantity space can be constructed using the landmark values $-\infty$, 0 and $+\infty$. This quantity space has five legal values: exactly $-\infty$, between $-\infty$ and 0, exactly 0, between 0 and $+\infty$, and exactly $+\infty$. Figure 2.7 illustrates the landmarks and regions used in this example. A qualitative variable also has an associated qualitative derivative that represents its rate of change. The qualitative derivative is also expressed using a quantity space which normally corresponds to the values: 'increasing', 'decreasing' or 'steady'.

The 'U-tube' system shown in Figure 2.8 is a common example used to illustrate qualitative representation [22]. Two tanks containing water, 'A' and 'B', are connected by a tube allowing water to flow between the tanks. The system includes a number of qualitative variables and associated quantity spaces. The flow between the tanks is modelled with a quantity space using the landmarks $-\infty$, 0 and $+\infty$. The pressure in a tank is modelled using the landmarks 0 and $+\infty$. The amount of liquid in A has the landmarks 0 and $maxCapacityA$, similarly 0 and $maxCapacityB$ are used for the amount of liquid in B.

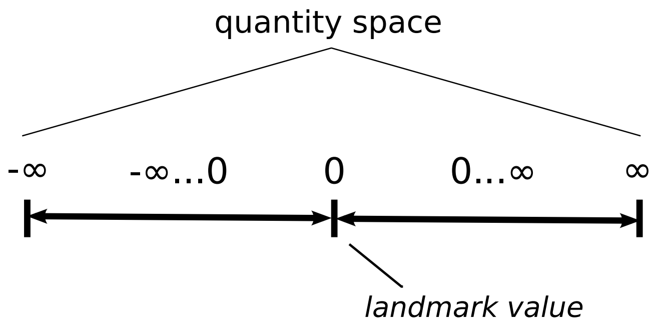


Figure 2.7: A quantity space with 3 landmarks and 5 legal values.

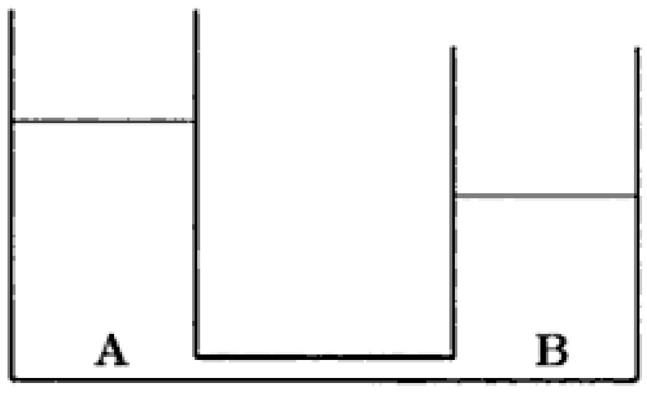


Figure 2.8: The 'U-tube' system.

Qualitative relations are used to model constraints between qualitative variables. These relations represent the types of constraints that would normally hold between quantitative variables, but they are interpreted in qualitative terms. Two important types of qualitative relation are qualitative proportionalities and qualitative influences. A qualitative proportionality between two qualitative variables specifies that they monotonically change with respect to each other. That is, if the real value of one variable increases, then the other always decreases or always increases. A qualitative proportionality does not specify by how much the other value changes — the relationship may be linear, polynomial or some arbitrary function, so long as monotonicity is maintained. In the 'U-tube' example, a negative qualitative proportionality can be specified between the variables representing the amount of liquid in each tank. It states that if the amount of water in tank A increases then the amount of water in tank B will decrease and vice versa. It does not state by how much the water in either tank changes. This rule applies whatever the shape of the tank. Other qualitative proportionalities could be specified for the U-tube system, including: between the amount of water in a tank and the pressure at the point where the tube meets the tank; and between the pressure differential and the flow rate between the tanks.

The other important type of qualitative relation is qualitative influence. Qualitative influences are used to represent what can cause a quantity to change. Influences can be positive or negative depending on whether they cause a quantity to increase or decrease. A variable is a positive influence on another if a non-zero value of the former causes the later to increase (given the absence of any other active influences), and vice versa. The state of an influenced variable can be determined by inspecting all of its influences: if they are all positive then the variable is increasing. If both positive and negative influences are active then the state is ambiguous: it may be increasing, decreasing or steady (the influences cannot be added since their quantitative values are not known). In the U-tube example, the amount of flow from A to B could be represented as a positive influence on the amount of water in tank B.

A qualitative process or system is a defined set of qualitative variables and constraints. A process is used to model the behaviour of an isolated set of objects. The process is not influenced by any outside conditions unless specifically allowed. A process is stateful, where a state is an assignment of each qualitative variable to a value.

The behaviour of a qualitative process or system can be discovered through simulation. A simulation starts with an initial assignment of variables and proceeds by adding all possible transitions to new states, with respect to the system constraints. This process is iterated until there are no further new transitions. The resulting behaviour graph can be used to predict the possible qualitative behaviours of a system. Properties of simulation algorithms vary depending on the types of constraints allowed. The QSIM simulation [23] for example is guaranteed to find all possible behaviours of a given system, however it is *not* guaranteed that all behaviours it predicts are valid. Tractability is also a problem given that the number of states is exponential in the number of variables. An 'envisionment' of a process is a directed graph of all reachable states and transitions.

2.3.2 Spatial Representation

Qualitative spatial reasoning is the sub-field of qualitative reasoning concerned with representing and reasoning about how objects occupy space and how they are spatially related to each other. Spatial reasoning can be applied in up to 3 dimensions depending on the problem domain. Surveys of qualitative spatial representations [8] [37], show that the 2-dimensional case is the most developed; the 3-dimensional case is usually considered as an orthogonal extension to a 2D representation if it is considered at all. Finding a useful general purpose spatial representation is currently an unsolved problem, however various representations have been successfully

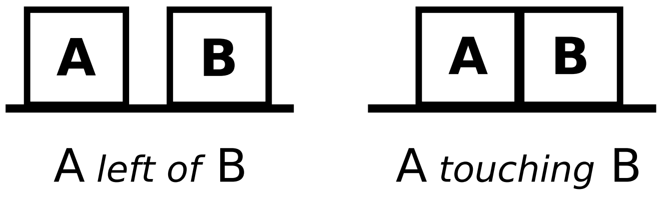


Figure 2.9: Example spatial relations used in the rectangle calculus.

applied to specific problem domains.

The rectangle algebra [19] is a typical qualitative spatial calculus. Its representation extends a 1-dimensional temporal representation based on intervals (Allen's 'Interval Calculus' [1]) to a domain with rectangular objects. Rectangles are specified according to their relationships with other rectangles. For example, rectangle A can be specified as *left of* rectangle B, or alternatively A could be *attached to* B. These relationships are illustrated in figure 2.9. Given a group of objects and relationships, further relationships can be inferred transitively, for example, if it is known that A is *left of* B and B is *left of* C, then A is *left of* C can be inferred. To represent the same situations quantitatively, a significant amount of extra information is required: a co-ordinate system to locate the rectangles absolutely, and precise co-ordinates for the corner of each rectangle.

It is believed that there is no general purpose (i.e. problem independent) purely qualitative spatial representation; this is known as the 'Poverty Conjecture' [16]. Forbus et al. argue that at some level of description, numerical values are required to fully predict the spatial interaction of objects. They site an example of two wheels rolling against each other, one with a 'notch' and the other a 'bump'; knowing whether or not the two wheels will roll smoothly without catching requires numeric information about the size and position of the irregular surfaces. They argue the solution to this problem is to ground the qualitative description 'place vocabulary' in a low level quantitative description 'metric diagram'. Ambiguities at the qualitative level would be resolved by reference to the quantitative values.

2.3.3 Actions in Qualitative Systems

Qualitative simulation predicts the possible behaviours of a system according to changes inferred from qualitative relationships between the objects. It does not consider changes that occur because of actions affecting objects within the system. For many problem domains such as robot planning it would be desirable to model both qualitative behaviour and actions simultaneously. Two possible approaches to integrating actions with qualitative processes have been proposed: *domain compilation* [21], in which the qualitative knowledge is compiled into a form for use with temporal planning systems; and *action-augmented envisionment* [15], in which qualitative envisionments are allowed to include states in which the fixed background assumptions are allowed to change.

Domain compilation involves creating planning rules from qualitative constraints. The compiled rules can be used in conjunction with action operators to predict new world states. The action operator predicts what will change, while the qualitative rules predict the indirect consequences of the changes according to qualitative constraints. A qualitative rule has antecedents and consequents that work in a similar way to action preconditions and effects (i.e. the antecedents must be true for the qualitative constraint to apply; the consequents will become true when the constraint is applied). The following rule taken from [21] is an example of a compiled qualitative constraint:

```
RULE
  Antecedents:
    (CONTAINED_LIQUID ?X) $c1
    (INCREASING (AMOUNT-OF ?X) ?CAUSE) $inc
Temporal Conditions:
    Exists (INTERSECTION $c1 $inc) called $int
Consequents:
    (INCREASING (LEVEL ?X) ?CAUSE) $int
```

The rule specifies that the *level* of a contained liquid is rising if something is causing the *amount* of liquid to increase. Action compilation allows qualitative processes to be used in planning algorithms, however, a significant problem with this approach is that interacting influences can lead to contradictions. For example, a contradiction is created when a container has a simultaneous inflow and outflow (such as when a sink has water flowing in from a tap and out from the plug hole). Qualitative simulation does not have this problem since ambiguous states are allowed.

Action-augmented envisionment takes the complementary approach to domain compilation by allowing qualitative models to include STRIPS like actions. Normally, a qualitative process requires a number of assumptions to hold for the predicted behaviour to be valid. An action-augmented envisionment allows actions to change the value of these assumptions. A new envisionment can be constructed that includes transitions that are the result of actions as well as the usual transitions caused by qualitative constraints.

A draw back of including actions is that construction of complete envisionment is difficult for non-trivial domains. The envisionment also requires simplifying assumptions such that actions only ever occur singly and that actions never coincide with qualitative transitions.

2.4 Learning Qualitative Models

A variety of systems have been developed that apply machine learning methods to the task of learning qualitative models. There are two main learning problems associated with learning qualitative models. The first problem is that of learning an appropriate variable quantisation from quantitative example data. This involves learning the real valued numbers that correspond to landmark values for a given quantity space. A good learning algorithm will find the landmarks that distinguish between interesting qualitative behaviours, for a given domain. Poorly chosen landmark values can lead to redundant qualitative states (and therefore increase the amount of computation required to simulate and envision), overly general states (in which useful qualitative distinctions are missing), and inaccurate states (in which a system does not behave as the qualitative model predicts it should).

The second problem is that of finding the qualitative constraints that hold between qualitative variables. The constraints can be learned from either quantitative or qualitative examples. A good learning algorithm with find enough constraints such that the qualitative behaviour of the variables involved can be accurately predicted. Missing or incorrect constraints will cause incorrect states to be found during simulation or envisionment.

The remainder of this section describes a number of exemplar systems for learning qualitative models. It includes systems that address both of the main learning problems described previously — learning landmark values from quantitative data, and learning constraints from quantitative and/or qualitative data.

2.4.1 Robotic arm control in a 3D simulated world

Mugan and Kuipers [29] describe a simulated 'robot baby' that learns qualitative rules for controlling its arm. This research is particularly relevant to this thesis because it addresses a similar problem, that of learning qualitative action outcomes within a 3D simulated environment with realistic physics. However it differs significantly in its objectives because it does not attempt to learn high level models of complex mechanisms. It instead focuses on learning a low level skill, that of moving the robot's arm, and does so in an extremely simple world that contains only the arm, a table and a block. The learning system does not assume that such basic skills have already been learned.

The input to the robot's learning algorithm is a time sequenced observation trace of a number of real valued variables. The variables represent the exact position of the arm, the position of the block, the closeness of the arm to the block, and the amount of power applied to the motors controlling the arm. It is assumed that the agent has a sensory system capable of observing the objects in this manner. Its vision system provides individuation of objects, tracking of objects and quantitative descriptions of the objects.

The learning mechanism attempts to find rules that reliably predict qualitative changes in state between variables. This includes creating landmark values to improve the reliability of the rules. A typical learned rule is written as:

$$\text{motorX}\rightarrow(302.23, +\infty) \Rightarrow \text{arm\_speedX}\rightarrow(0, +\infty)$$

This rule specifies that the qualitative state on the right side of the expression will reliably occur soon after the state on the left side has occurred. This particular example rule can be read as: "activating 'motorX' with power greater than 302.80 will cause the arm to move in a positive direction along the x-axis". The rules are learned by first creating candidate rules based on changes in qualitative state that have a high likelihood of occurring together in the observation traces. The candidate rule is then refined by introducing landmark values and context variables that specialise the rule and increase its observed reliability.

The algorithm explores some interesting ideas: using closely occurring qualitative state changes to infer a causal relationship; making a distinction between causal (i.e. 'happens soon after') and simultaneous state change rules; and creating new landmarks to increase the reliability of predictive rules. These are applied to create an agent which can learn accurate descriptions of low-level qualitative actions. Some of these ideas may be applicable to learning action models in a more complex world, however, the current system assumes a very limited environment with just a small set of observables.

2.4.2 Predicting qualitative physical interactions

Boxer [5] describes a system that learns to predict qualitative state changes in a simulated environment with realistic physics. The goal of the system is to learn three specific naive physics rules by observing the movement of objects in a 2D world. The world is simple and contains only a wall (stationary) and some billiard balls that move and collide. It aims to learn a model that captures the following common-sense knowledge: 'objects are solid', 'objects must touch to affect each other' and 'objects move along continuous paths'. This is the kind of knowledge that an agent would require to act intelligently in a variety of physical environments.

A simple qualitative spatial representation is used to represent the world states. Objects are observed in terms of their qualitative spatial relationship to each other. For example, a ball A may be to the left of another ball B, and at the same time moving to the right (i.e. away from B) at some speed greater than 0. State transition probabilities are represented using a Bayesian Network, figure 2.10 shows the structure of the network. Nodes in the network represent the current object ('reference object'), a second object ('located object') and the current spatial relationship between them; the dependent node represents the next spatial relationship. Learning is achieved by updating the network with changes in qualitative state be-

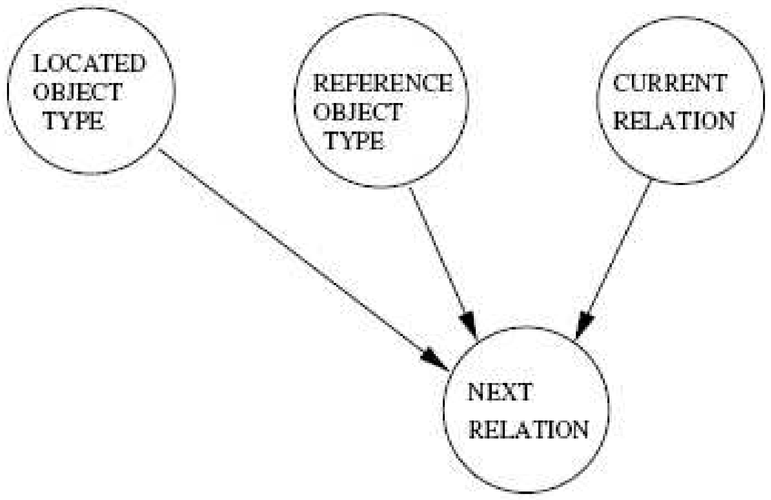


Figure 2.10: Bayesian network for qualitative state changes.

tween adjacent time frames. After enough trials the network is able to accurately predict the 'next relation' of the objects such as balls moving away from each other after a collision. The system functioned well in simple trials but was unable to cope with a trial containing multiple balls and interactions.

This system demonstrates that it is possible to learn simple qualitative state changes using a Bayesian network in a simulated physical environment. However, the system only worked when interactions were easily isolated and involved only two objects. It is not clear how such an approach could be scaled up to be useful in a more complex environment.

2.4.3 QUIN: inducing a qualitative decision tree

The final system in this section learns qualitative rules from numeric data. The rules are used in a software system that controls physical processes. The motivation for the research comes from a requirement to reverse engineer the control mechanism used by a gantry crane. The crane has a control mechanism that safely lifts heavy objects to and from trucks. The

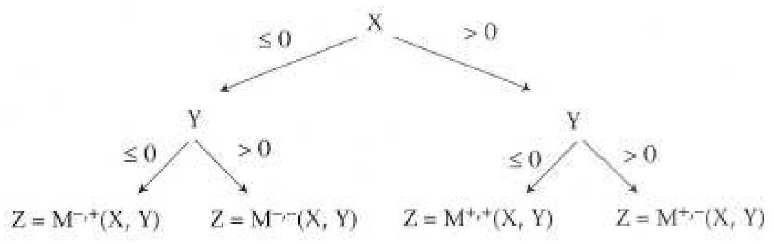


Figure 2.11: Example learned qualitative tree.

learning algorithm is called QUIN [6] (short for QUalitative INduction). QUIN creates 'qualitative trees' that are similar to decision trees but have qualitative constraints at their leaves. Figure 2.11 shows the output from QUIN for a noisy data set of points {X, Y, Z} where $Z \approx X^2 - Y^2$. The tree in figure 2.11 shows how Z is qualitatively constrained when X and Y are in particular qualitative states. For example, when X is positive and Y is negative the constraint '$Z = M^{+,+}(X, Y)$' holds. This constraint states that Z monotonically increases in both X and Y.

The algorithm learns in a similar manner to normal decision tree induction. However, the qualitative constraints (that take the role of classes) and discriminating thresholds are not fixed beforehand. When splitting on a node, all possible thresholds and constraints are considered. Constraint/threshold combinations are selected by searching for those that are most consistent with the observed data.

QUIN is useful for finding the qualitative relationship between variables in situations where finding a precise quantitative function is not possible. This type of approach could be used in conjunction with a relational decision tree learner (the TILDE [4] system for example) to learn qualitatve models of relational domains. However the problem of learning from unbounded problem domains would still have to be addressed.

2.4.4 INTHELEX: learning naive physics

Esposito et al. [12] describe the application of the INTHELEX learning system to the problem of learning naive physics. The goal of the system is to replicate the kind conceptual theory revision observed in small children. For example, in early childhood, the concept of 'force' is explained as an innate property of big or heavy objects (they are hard to move, so they have force), as the child develops the concept is revised as an acquired property of moving objects (moving objects are given force by the agent that set them in motion).

INTHELEX is an ILP engine that learns hierarchical concepts as logic programs. The system incrementally refines concepts by matching examples to existing concepts and then, if required, specializing in the case of negative examples and generalzing in the case of positive examples. The system retains all examples it has been shown and ensures that learned concepts are consistent with them all.

The system was provided with a number of positive and negative examples constructed from qualitative predicates such as $weight\_low$, $weight\_medium$, and $weight\_high$ (note that the qualitative values 'low', 'medium', and 'high' are arbitrarily chosen). The following is a typical training example:

$$\begin{aligned} has\_innate\_force(s,t) \text{ :- } & stone(s), size\_high(s), weight\_high(s) \\ & man(m), size\_high(m), weight\_high(m) \\ & stationary(s,t), stationary(m,t), \\ & pushes(m,s,t) \end{aligned}$$

It represents the concept that 'a large stone has *innate force* because it remains stationary after a large man pushes it'.

The system was able to learn general concepts for both innate and acquired force, which it is argued, are similar to those adopted by children. The following is an example learned rule:

$$\begin{aligned} has\_innate\_force(X,T) \text{ :- } & size\_high(X), weight\_high(X) \\ & stationary(Y,T), stationary(X,T), \end{aligned}$$

$$pushes(Y, X, T)$$

The approach gives some insight into how child-like learning of qualitative models could be used as a template for intelligent agent learning. However, the rules were learned from examples drawn from a very simple world in which only the relevant objects exist. Learning such rules in a more complex environment would be significantly more difficult. The system also assumes that the examples are noise free; that both positive and negative examples are avaible from which to learn; and that perfect qualitative landmarks have been discovered.

Chapter 3

Q-Systems

Knowledge representation is a key component of an intelligent agent system. The representation must be sufficiently expressive to enable the agent to represent, and to reason with, the knowledge it needs to achieve its goals. It is therefore highly dependent on the types of tasks the agent is required to achieve and the types of environments the agent will operate in. An agent that must operate in a wide variety of unforeseen environments requires a more expressive representation than an agent which operates in a highly constrained environment; similarly a more expressive representation is required if an agent is to describe and achieve arbitrary goals rather than some pre-determined set of tasks.

The choice of representation will not only affect the agent's operational effectiveness but also its ability to learn new knowledge. A more expressive representation can be more difficult to learn because it includes more types of things and describes them in greater detail. A good representation balances the level of description (expressiveness) with the ability of the agent to learn (learnability). Furthermore, an overly expressive representation is not only detrimental to learning but can also lead to overly detailed models of behaviour which can make the task of planning more difficult.

This chapter describes the 'Q-System' representation, a novel representation which describes 'systems' of interacting objects. The representation is designed to enable an agent to operate at a human ('common sense') level in arbitrary everyday environments. Q-Systems are based on non-deterministic finite state machines and aim to combine an expressive action representation with qualitative process modelling. The resulting integrated representation can describe both action effects and qualitative behaviour. The following sections describe the design goals, the representation details, a Q-System notation, and finally a discussion of the important design decisions.

3.1 Goals

The representation has several goals that follow from the overall goals of this research. The primary goal of the representation is to enable the agent to learn models that can be used to achieve arbitrary tasks in a wide variety of everyday environments. Therefore the representation must be independent of any single problem domain and expressive enough to model arbitrary unforeseen situations. A key idea is to partition the world into independent systems of interacting objects, and this must be supported by the representation.

To achieve these goals a number of desirable features are considered essential characteristics of a successful representation. Conversely, a number of limitations are tolerated, either due to practical reasons or to constrain the scope of the research. The essential features and tolerable limiations are related to three distinct areas of the representation: those that apply to the representation of world state, those that apply to representation of state change, and those that apply to the representation as a whole. The features and limitations applicable to the representation of world state are:

- The state of the world should be represented in terms of objects and discrete properties. It is assumed that a vision system has already determined what objects exist and can observe their properties and relationship to each other.

- It should be possible to model arbitrary information about objects. This follows from the goal of learning in arbitrary environments. The representation should not be limited to a specific set of observable features derived from a single problem ddomain. The representation cannot make any assumptions about the specific features that objects may have.

- Real valued variables should be represented qualitatively. This follows from the assumption that real valued variables cannot be directly observed and that qualitative measurements are used in their place. It is a goal of this research to show that qualitative descriptions are sufficient for everyday problem domains.

- The representation should support descriptions of generalized models which can be matched to specific situations. Therefore the representation must have some way of referring to generalized objects using variables.

- The representation should allow explicit representation of groups of objects. It is assumed that the vision system can distinguish interesting groups of objects. Groups of objects also have properties and can behave as a single object.

- The representation should support representation of partial world states. A goal of this project is to learn models in an 'open world'. It is unrealistic to assume that world states are completely observable and therefore the representation should support partial states.

The features and limitations applicable to the representation of state change are:

- The representation should represent qualitative world dynamics, i.e. how the qualitative state of the world changes under different conditions (especially with regard to the agent's actions). This will allow the agent to plan by calculating future states that satisfy its goals.

- Actions will be represented atomically in terms of how states change when they are executed. This limitation is a simplifying assumption that allows the *effects* of actions to be the primary focus of learning rather than their *execution*.

- The distinction between changes resulting directly from actions and changes resulting from world dynamics should be represented explicitly. To plan effectively an agent must have knowledge of how the world changes 'by itself' without intervention from the agent.

- Observable non-qualitative changes in real valued variables will be represented. Certain changes in the environment will not result in a change of state because real valued variables are abstracted into discrete quantity spaces. However, these changes can be important to understanding a process and, since they can often be observed, should be explicitly representable as effects of actions.

- The probability of a given state transition will be represented. A given action may result in several different outcomes; probabilities on the various outcomes will allow the agent to plan more effectively.

The features and limitations applicable to the representation as a whole are:

- Isolated groups of objects interacting together as a system should be explicitly represented. This follows directly from one of the overall project goals which is to investigate the usefulness of such partitioning.

- The representation should be optimized for on-line learning from small numbers of examples. This limits the scope of the research to a particular type of autonomous agent architecture. These limitations are realistic for an agent that must learn quickly in new environments.

3.2 The Q-System Representation

The Q-System representation has been designed to fulfil the goals outlined in the previous section. The representation defines 'systems' which are finite state machines representing the qualitative behaviour of a group of related objects. The representation builds on ideas from qualitative reasoning and action description languages by integrating an expressive action description language with a qualitative behaviour graph (also known as an envisionment). The remainder of this section describes the components of the representation in detail.

3.2.1 Objects

The representation models the world in terms of objects. An object is a part of the world that has been identified as an object by the vision system. Objects are uniquely identified, exclusively occupy space, are described by their properties and the context of their relationships with other objects. Over time objects can change as they interact with other objects. It is the job of the vision system to identify such objects and track them over time (determining what exactly is and is not an object is non-trivial and is discussed in following sections).

Objects can be either individuals or compositions of other objects. The representation puts no constraints on how object composition hierarchies are organized. This means scenes can be observed in different ways depending on the particular compositions used. For example, the vision system may perceive a chair as an individual object, or alternatively, it could perceive the individual component objects (wheels, base, seat and back; but not the chair as an object itself), and finally, it could perceive both the component objects and the chair as a composition of them. It is assumed that the vision system can identify and construct an appropriate level of composition for a given scene.

Properties describe objects and have a discrete range of possible values. Exactly one value is assigned to each property of an object. For example, an object may have the property 'colour' and it may be assigned the value 'blue' (it cannot be assigned two values, say 'green' and 'blue' to represent turquoise, an explicit 'turquoise' value would be required to represent this). The actual properties for a given object are determined by the vision system.

Qualitative properties are special properties that describe real valued measures in discrete terms. Qualitative properties take two discrete values: a qualitative value from an ordered quantity space and a qualitative derivative describing the rate of change of the underlying real value. Every qualitative property has its own quantity space, for example, a qualitative height-of-water property of a cup object may have a quantity space containing values: [0, 0-FULL, FULL]. All qualitative derivatives are assigned a value from the range: [decreasing, steady, increasing] (or 'dec', 'std', 'inc' for short).

Relationships describe how two objects relate to each other. An object with properties is just an object but an object with relationships becomes part of a scene. The precise values of the relationships describe the context of the object within the scene. Relationships specified between two objects are assigned a discrete value. For example, if two objects are touching then the value of the 'touching' relationship between the objects is 'yes'. Relationships can take any value but are typically boolean. The relationships in which an object participates are determined by the vision system.

Relationships can be either symmetric or asymmetric. The relationship ‘on’ is asymmetric because X-on-Y is a different assertion to Y-on-X. However, X-touching-Y is a symmetric relationship because X-touching-Y implies Y-touching-X. Asymmetric relationships may be constrained, for example, X-on-Y implies that Y-on-X is false.

Groups of Objects

The representation explicitly represents special groups as a special kind of composite object. Group objects are sets of objects that are grouped according to some identified common property. Group objects have properties and relationships just like normal objects. An example of a group is a stack of boxes. The defining common property of the member objects is that they are of the same type and are vertically aligned. The stack of boxes may have a qualitative height property and/or relationships with surrounding objects. The stack may have the top box and bottom box identified as ‘special’ member objects. It is assumed that the vision system identifies group objects, either by noticing shared properties and relationships or when trying to match a scene to a description of a group object.

3.2.2 States

A ‘state’ in a Q-System is a conjunction of assertions about objects. The assertions represent a particular configuration of objects, for example it might describe the part of the world observable to the agent at a given point in time. An assertion is an assignment of a value to a fluent (i.e. object property or relationship). If a particular fluent is not assigned any value in a state then the value is either unknown or unspecified depending on the context in which the state is being used. States used in system descriptions use unassigned fluents to indicate ‘unspecified’ in the sense that the state is compatible with a world state in which the fluent can take any value. On the other hand, world states observed by the vision system use unassigned fluents to indicate ‘unknown’ in the sense that the value of the fluent cannot be observed at this time.

Observed world states are normally partial states. They do not include all assertions that are true of the world when it was observed. This is a limitation of a realistic vision system. It is not possible to always observe every property of every observable object.

3.2.3 State Transitions

Transitions represent how states change over time, either due to an agent's action or the underlying world dynamics. Actions are represented by a type identifier that uniquely identifies the type of action (e.g. put-on), the target objects on which the action is executed, a frequency count, a set of qualitative deltas, and finally a 'pre' and one or more 'post' states. The post-states of an action represent the possible outcomes of the action when it is executed in the specified pre-state. The frequency count of a transition represents the number of times this particular transition has been observed. It can be used as a rough probability estimate of the actual post state when a given action has a set of different possible outcomes.

A transition's qualitative deltas represent changes to qualitative properties that are observable but not significant enough to result in a qualitative change of state. For example, the water level in a cup may increase but does not change enough to be move into a new qualitative state (e.g. from '0-FULL' to 'FULL'). In this case the property's *delta* is given the value 'increased' for the transition. Qualitative deltas take the values: 'increased', 'not changed', or 'decreased'. Qualitative deltas, like actions, are *not* part of the state because they are transient. They are observed to have happened since the last observation but say nothing about the current state of the world as they are not currently happening.

Non-actions are represented in the same way as actions except they are assigned a special 'time-passes' action type and do not have target objects. Time-passes transitions represent world dynamics in action. For example, when a falling object hits the floor it stops moving and the state of the world changes. The observed transition from moving object to stationary object on the floor can be represented as a time-passes transition.

A transition does not specify a length of time over which the transition occurs. The transition represents an arbitrary amount of time sufficient for the transition to occur. This is assumed to be a short amount of time (determined by the vision system), typically under a second, however certain actions may take longer in which case the representation allows for them to be consistent with the model. Enforcing a fixed time interval would require modelling durative actions which is beyond the scope of this project as actions are assumed to be atomic.

3.2.4 Systems

Systems of interacting objects are represented using a Q-System. A Q-System consists of two parts: a context and a behaviour. A system's context defines what must be true of a set of objects for the system to be applicable. For example, the system in figure 3.1 requires a spout above a sink with a tap and plug controls to be applicable. The context is the precondition of the system and is true in all the system's states. A system's behaviour is a finite state machine describing the possible states that the system can be in and the possible transitions between states. The system, if applicable, is in one of the states described in its behaviour. State transitions are either actions or time-passes transitions as described previously. The 'scope' of a system is the set of objects described in the system's context.

A system's action transitions only describe the effects on the system's objects. Objects outside of the system may be affected but the explicit effects are not defined in the system. It is assumed that the effects are not important to the working of the system at hand.

Figure 3.1 shows the behaviour graph of an example tap and sink system modelled using the Q-System representation. The system involves several objects, a tap, a sink, a plug and an handle. The system is a model of a real tap and sink which is controlled using the handle and plug objects. Rotating the handle left or right affects the flow of water coming from the tap. The plug object controls the drain: inserting it will close the drain to prevent water escaping; removing it will open the drain allowing water to leave the sink. The system has several states which can be explored by operating the controls and allowing the water to fill and drain away. Individual states are described using the qualitative properties of the tap and sink objects, their water-height, flow-rate etc. (Note that not all possible states and transitions are modelled, for example inserting the plug before the water has fully drained).

The tap and sink system's context describes the required components and their configuration. It requires that the objects must be the appropriate type, that the tap must be located above the sink, etc. The context of the example system is:

$$\begin{aligned}
W.isa &= plug \\
X.isa &= tap \\
Y.isa &= handle \\
Z.isa &= sink \\
Z.hasDrain &= yes \\
X.connected.Y &= yes \\
X.above.Z &= yes \\
X.unobstructedPath.Z &= yes
\end{aligned}$$

Note that in this particular context all the objects are defined as variables. This need not necessarily be the case, specific objects may also be identified in the context. Table 3.1 lists some of the transitions of the tap and sink system (the seven transitions involving states *a*, *b*, *c*, and *d*).

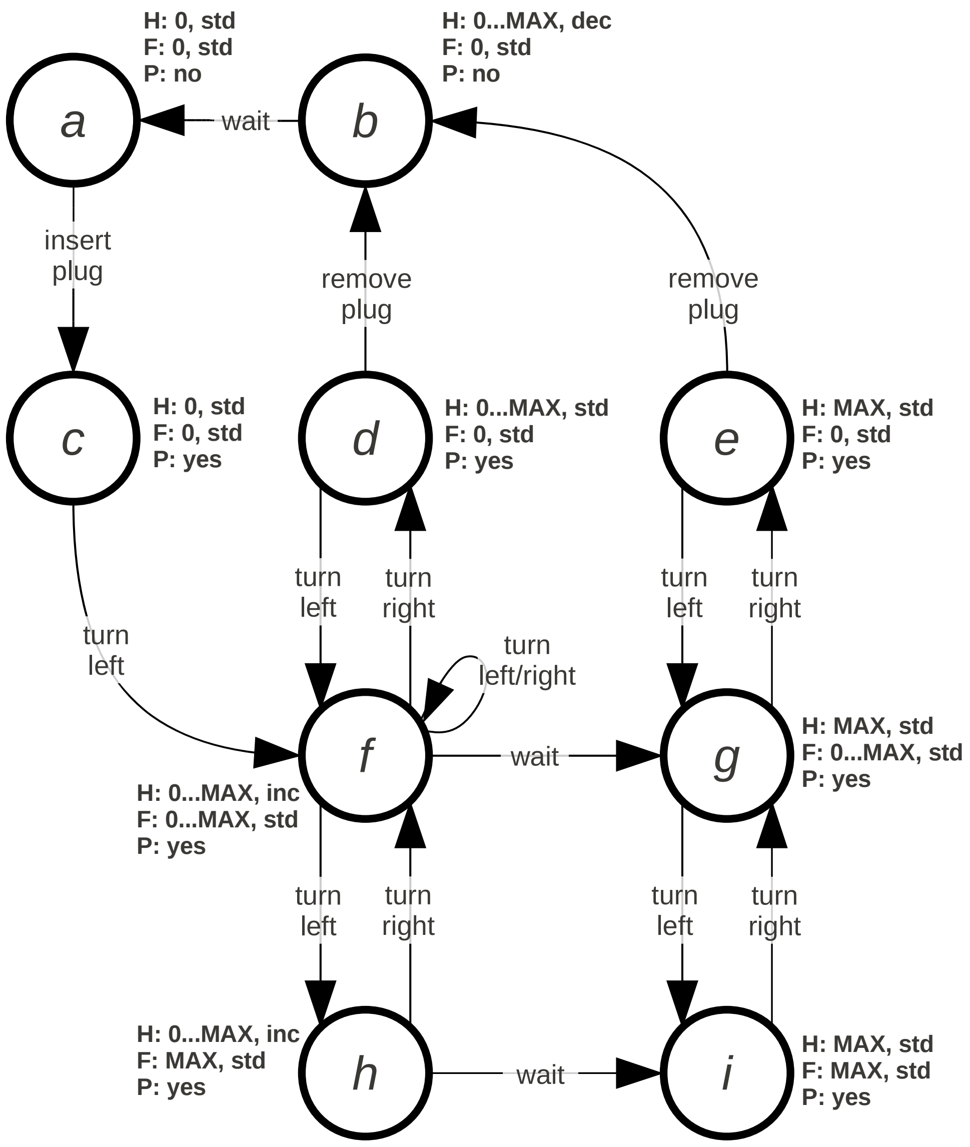


Figure 3.1: Part of Behaviour Graph for Simple Sink System. The values for **H**, **F**, and **P** correspond to the variables water-height, flow-rate and plug-in respectively. The sink is empty in states *a* and *c*; the sink contains water in states *d* and *e*; the sink is filling in states *f* and *h*; the sink is overflowing in states *g* and *i*; and the sink is draining in state *b*.

	pre-state	**action**	**post-state**	**deltas**
height:	0...MAX, dec	*wait*	0, std	dec
flow-rate:	0, std		0, std	
plug-in:	no		no	
height:	0, std	*insert-plug*	0, std	
flow-rate:	0, std		0, std	
plug-in:	no		yes	
height:	0...MAX, std	*remove-plug*	0...MAX, dec	
flow-rate:	0, std		0, std	
plug-in:	yes		no	
height:	MAX, std	*remove-plug*	0...MAX, dec	dec
flow-rate:	0, std		0, std	
plug-in:	yes		no	
height:	0, std	*turn-left*	0...MAX, inc	inc
flow-rate:	0, std		0...MAX, std	inc
plug-in:	yes		yes	
height:	0...MAX, std	*turn-left*	0...MAX, inc	inc
flow-rate:	0, std		0...MAX, std	inc
plug-in:	yes		yes	
height:	0...MAX, inc	*turn-right*	0...MAX, std	
flow-rate:	0...MAX, std		0, std	dec
plug-in:	yes		yes	

Table 3.1: Example Tap and Sink System Transitions

Some actions have no qualitative effect on the state of the system (e.g. rotating the handle when the tap is already on, see state f in the diagram). They do however have a qualitative delta annotation indicating that there was actually some change (e.g. the flow-rate changed), just not a qualitatively observable change. Notice also that the system has non-deterministic actions because there is insufficient information in the qualitative state to determine the outcome of the action (e.g. again, rotating the handle when the tap is on, see state f).

3.3 Notation

A simple short-hand notation is used to refer the components of a Q-System. The remaining chapters use the notation described below.

A Q-System is a three-tuple describing its context $s_{context}$, the set of system states S and the set of transitions of the system T:

$$system \quad = \quad < s_{context}, S, T >$$

Transitions are five-tuples describing the before state $s_{pre} \in S$, the action type a_{type}, the list of action targets $a_{targets}$, the after state $s_{post} \in S$, and the set of qualitative deltas D:

$$transition \quad = \quad < s_{pre}, a_{type}, a_{targets}, s_{post}, D >$$

A common transformation is to substitute variables for object identifiers. This is indicated using the '/' operator. The following example denotes that the a new system has been defined by applying the variable substitution sub to the system $system$:

$$newSystem \quad = \quad system/sub$$

Variables are distinguished from object identifiers by using uppercase letters. So 'X' denotes a variable and '$obj1$' denotes an actual object.

3.4 Design Considerations

This section discusses the important considerations taken into account when designing the representation.

What is a System?

The Q-System representation uses the notion of a system which is a group of interrelated objects with a behaviour. The intuition behind this abstraction is that knowledge about how the world works is more useful and can be learned more easily when it is modularized; organizing knowledge as systems is a type of modularization. Modularization allows knowledge to be learned more easily because the number of objects operating in the system is restricted, there are far fewer interactions to explain. The modularized knowledge can be more useful because the scope of the world is limited to relevant objects. For example, a system encapsulates some planning knowledge simply by discarding many possible but irrelevant consequences. If a system happens to contain both the start and goal state in a planning problem, then planning becomes a trivial task of navigating the system.

Systems can vary in scope considerably. The simplest 'system' is just a single transition (which is equivalent to a single action description rule) but systems can be arbitrarily complex. An aim of this research is to find a good heuristic for deciding what is the most useful size for a given system. I.e. when should objects and transitions be included and when should they be omitted?

Intuitively, systems would appear to be a natural way of usefully storing knowledge and it is the purpose of the learning system to show that this is the case.

Extending Action Description Languages

The Q-System representation uses ideas from action description languages such as STRIPS [13]. Q-Systems describe actions in terms of a pre-state, which is like a STRIPS precondition, and a post-state, which is similar to STRIPS add and delete lists. Furthermore, each transition in a Q-System has a frequency count which is similar to probabilistic action rules (such as those used by Zettlemoyer et al. [45]).

Despite these similarities there are some key differences between Q-Systems and action description type languages. A key difference is that systems allow action pre-conditions to be learned within the scope of a group of objects with interrelated behaviour. This is in contrast to action description languages in which pre-conditions of an action are learned in an unrestricted scope. This means that revising an action involves considering all possible applications of a particular action. Consider the very generic 'push' action: there are a huge number of possible situations in which the action applies and a similarly large number of possible outcomes. Learning the action as part of a system restricts the possible situations and outcomes to only those involving the objects of the particular system. The system approach also allows a group of actions to have a shared pre-condition via the system context. This means that revising the system context will revise the pre-conditions of a number of actions simultaneously. This can be more efficient than learning them all individually.

Q-Systems also include the addition of qualitative deltas to each transition. The qualitative delta provides extra information about the effects of an action that cannot be captured in the (qualitative) state description. The deltas were added to solve the problem of actions apparently not having any effect when the effect was too small to constitute an observable qualitative change. Q-Systems also differ from other action languages in that they explicitly represent qualitative variables. This allows the learning and planning systems to make use of the special properties of qualitative variables. For example, the quantity spaces used by qualitative variables are ordered and variables must transition through each possible value in order. This information can be used to make inferences about incomplete observations for example.

Another difference between Q-Systems and action description languages is that world dynamics and action effects are represented in the same way (as state transitions). This is in contrast to action description languages which focus only on actions and rely on world dynamics being represented as part of the 'domain knowledge'. This leads to actions and world dynamics being treated as two distinct learning problems. The integrated Q-System representation is specifically designed to allow both to be learned simultaneously.

Black Box Vision System

The Q-System representation has been designed with the assumption of a sophisticated vision system that provides an object orientated view of the world. The vision system provides a service which supplies regular snapshots of the world to the learning system. The learning system requires no knowledge of how the vision system works (it is a 'black box'). This separation reflects an architectural decision to keep the vision and 'high-level' learning systems distinct and focus on the learning system alone. This restricts the scope and difficulty of the problem addressed by this thesis.

The obvious alternative architecture is based on integrated vision and learning systems in which the functionality of the vision system and high-level learning system interact. In such an architecture the task of learning a high-level model of how a toaster (for example) behaves may be performed simultaneously with the task of determining exactly what constitutes a toaster. The representation used by an integrated architecture would potentially need to be much richer than the type of representation built with a black box vision system in mind. So to a certain extent the design of the Q-System representation can be simpler because of the architectural decision to treat the vision system as a black box.

Lack of Quantitative Variables

The representation does not include quantitative variables. This decision was made in order to explore the power of qualitative abstraction over quantitative modelling and investigate questions such as: Is it possible to learn sufficiently useful models *without* quantitative information? This restriction is somewhat artificial because a vision system that provides relatively accurate quantitative measurements is quite possible. However, there are a variety of circumstances in which qualitative abstraction is required which means it cannot be assumed that quantitative information is *always* available. For example, it is difficult to estimate the speed of an object that is moving towards the viewer. A qualitative observation that the object is moving at some speed greater than zero is easier to observe.

The lack of quantitative variables and formulas means that the representation is unable to model certain behaviours as accurately. This sacrifice of expressivity has been made in order to reduce the scope of the research problem. It would be more challenging to learn both qualitative and quantitative models and also determine when each is appropriate for a given process. Ideally an agent would have both types of representation available for building models of observed behaviour.

Object Orientated State Descriptions

The representation describes world states in terms of discrete objects in which every object is unique and indivisible.[1] This is a simplifying assumption as it forces a partitioning of the world into objects when it is not always obvious how to do so. A tree for example can be viewed in a variety of different ways and using different objects, depending on the level of detail required. The tree can viewed as a single tree object, or a combination of trunk and branch objects, or trunk and branch and leaf

[1]Indivisible in the sense that if an object (such as a plate) is broken then the resulting parts become new objects.

objects, etc... This can be taken to the extreme by considering individual cells as objects, which maybe an appropriate level of detail if analyzing the chemical interactions within the tree.

A richer representation would include a more flexible notion of object that can more accurately describe the tricky situations where object partitioning is not obvious. Such a representation might include hierarchies of objects. A hierarchy would allow objects to be viewed simultaneously at multiple levels of detail. Another strategy is to relax the discreteness requirement on objects. For example, consider an oil slick in the ocean, a certain area of the oil slick could be said to be 90% oil slick 'object' and 10% ocean 'object'. This would enable reasoning about different areas *within* the defined bounds of an object.

Addressing these issues is considered outside the scope of this project. It is assumed that the vision system has partitioned the world into appropriate objects and the representation reflects this.

Chapter 4

Simulation, Vision System & Skill System

Ideally the learning system presented in this thesis would be developed, tested and evaluated using a physically embodied agent with the ability to interact with and observe the physical world. Unfortunately, such an agent with the required high-level symbolic vision system does not yet exist and it would be a major undertaking to develop one from scratch. Therefore the only feasible alternative is to use a simulated world and an agent with simulated vision and skill systems. Learning algorithms are developed and assessed within the simulation which acts as substitute for the physical world. This chapter describes the goals and implementation details of the simulation, skill and vision systems that were created for this thesis.

The simulation is designed to be complex enough such that many challenging real world learning problems can be replicated. The following properties of the real world contribute to the difficulty of learning about the world, and therefore it is important that the simulated world shares these properties:

Intractability, the simulation should be intractable in the sense that an agent could not learn a complete model of its environment by exploring every possible state of the world and remembering everything about each state. In particular the world must contain enough objects such that exploring all possible states is not computationally feasible. The simulation must require the agent to perform filtering and generalisation in order to create good quality, useful models.

Stochasticity, the simulation should allow the agent to observe randomness in the interactions between objects. This means two sets of objects in the same state can be observed on different occasions transitioning to different states.[1] The outcomes for a given interaction are not arbitrary, they are governed by some probability distribution. (Note that stochasticity is distinct from partial observability which is a property of the vision of the system. In the later case, outcomes appear random because of missing information rather than underlying true non-determinism.)

Diversity, the simulation should allow the agent to learn in a diverse set of scenarios. A key goal of the agent is to perform domain-independent learning, therefore the simulation should not be constrained in the number and types of scenarios it can provide. The more diverse the environments in which the agent can be shown to learn successfully, the stronger it can be argued that the learning is generally applicable to learning in the real world.

Richness, the simulation should exhibit sufficient complexity in object interactions such that qualitative abstraction will be an advantage for an agent with limited vision capabilities and limited time to explore the environment. This supports one of the key goals of this research which is to demonstrate the usefulness of qualitative abstraction.

[1]Whether true randomness exists is a problem we will leave to the physicists. It is assumed that the resolution of agent's vision is sufficiently limited that apparently random events occur.

> The simulation should allow objects to interact in subtle and complex ways that would require non-trivial quantitative formula if they were to be modelled *precisely*.

A simulation exhibiting all of these properties presents a learning problem that is difficult in many of the same ways as the real world is difficult for physical agents.

The properties described so far are desirable in order to create a simulation of sufficient complexity, the following additional requirements allow for greater flexibility in the types of learning scenarios that can be investigated. Firstly, the simulation should support optionally turning on or off the various complexities described. This allows learning to be assessed under different conditions and also allows the effect of particular complexities to be isolated. Secondly, the simulation should support 'teaching' the agent. A human teacher should be able to execute actions within the simulation and have them observed by the agent. The actions performed by the teacher should be of the same type as those performed by the agent.

The goal of the agent's vision system is to describe the world at an abstract level in terms of objects in such a way such that it could plausibly be implemented in a physical agent. Describing the world at an abstract level involves partitioning the environment into discrete objects and uniquely identify them. The identified objects must then be described in terms of 'observables' that support creating models using the Q-System representation (see chapter 3). The vision system is shaped by the representation which dictates the types of observables that can be observed: simple properties, qualitative properties, relationships, etc. For each scenario or world that is created the vision system will implement a variety of instances of these types. For example in a world containing moving objects, the vision system may implement a 'speed-of-object' observable as an instance of a qualitative property.

It would be unrealistic to allow the vision system to have a complete view of every object in the world. Both humans and physical agents only ever see a small subset of the complete state of the world at any given time. Therefore the vision system is limited in that it is possible for objects (and individual properties) to be unobserved at a particular point in time and for them to potentially become observed at a later time. For example, putting an object into a cupboard hides it from view and it becomes temporarily unobserved. It is also possible for some objects/properties to be totally unobservable (i.e. they can never be directly observed). For example, the weight of an object can inferred in several ways but never directly observed (at least visually).

A realistic vision system is 'noisy' in the sense that it is possible for the agent's vision system to make incorrect observations (i.e. assert something is true about the world when it is not).

Finally the vision system must work in discrete time so that the world is observed as a series of discrete states progressing over time. This is simplifying assumption that supports modelling the world in terms of discrete state transitions. A real time system is plausible but would be more difficult to implement. Also the additional fine grained information would be of limited use to a qualitative vision system where changes in state are discrete (compared with a quantitative vision system where change is continuous).

The agent's skill system has a similar goal to the vision system. The goal of the skill system is to execute abstract actions that affect target 'objects' in such a way that they could be plausibly implemented in a physical agent. The agent's skills are assumed to be have been already learned by the agent (e.g. the agent can pick an object up, it does not need to learn the precise control over each finger required to do so successfully). The actions are learned sufficiently well such that they affect the world in a reliable way (although not perfectly reliably). Actions can involve multiple objects and the skill system supports an arbitrary number of target objects.

Actions are executed atomically. This is a simplifying assumption that allows the learning system to focus on learning the effects of actions rather

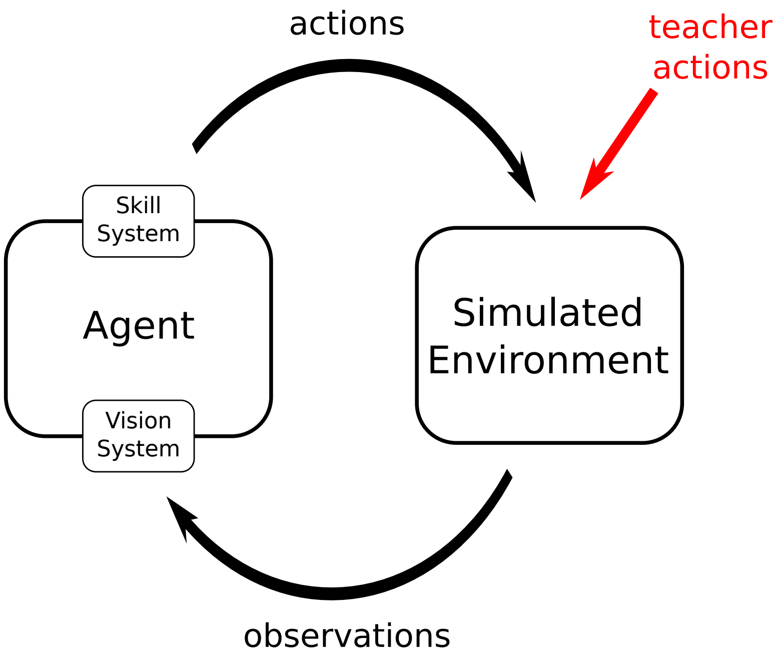


Figure 4.1: Conceptual Architecture

than their implementation. The set of actions available to the agent are preconfigured, this thesis does not attempt to address the problem of how the agent learns what actions it can execute or how they are executed. Finally, the agent's skill system is 'noisy', meaning it can incorrectly perform actions (i.e. does not change the world in the way it is meant to).

4.1 Experimental Setup

This section describes the conceptual components and software implementation of the complete agent system. The system provides a test bench on which to run experiments with learning algorithms and associated representations.

The diagram in figure 4.1 shows the conceptual architecture of the agent system. The system has the following components:

Agent The agent interacts with the simulation using its skill and vision systems. It receives high-level abstract observations from the vision system. It can request its skill system to execute a high-level abstract action. The agent implements a learning algorithm that constructs models that the agent can use to predict future states.

Simulation The simulation implements an environment with which the agent can interact. It receives low-level instructions from the agent's skill system to modify the current state of the world. It provides low-level object data to the agent's vision system. The simulation's state changes over time according to action executions and the particular world dynamics of the simulated environment.

Skill System The skill system transforms high-level abstract actions into low-level instructions that can be executed by the simulation to change the current world state. The skill system is not perfect and may incorrectly execute an action.

Vision System The vision system transforms low-level abstract data about simulation objects into high-level abstract observations. The vision system may introduce noise into the observations.

Teacher The teacher is a human operator who can interact with the simulation by executing actions in the same manner as the agent. The teacher is an optional component.

This architecture can be compared with a physical agent in which the simulation is replaced with the real world and the agent replaced by a robot. In this case the architecture could remain the same with the exception that the physical agent's skill and vision systems would be far more sophisticated. *If* such sub-systems existed then the algorithms internal to the physical agent could be identical to those used by the simulated agent. This would only be the case if the simulation used to develop the learning algorithm was sufficiently realistic, otherwise the learning may be significantly less effective.

Note that because the teacher is exogenous to the simulation (i.e. the teacher does not 'inhabit' the agent's environment) the agent perceives the actions of the teacher as though it had executed them itself. A more realistic architecture would include the teacher as part of the simulation, along with a mechanism for allowing the agent to observe the teacher's actions as 'special' for the purpose of learning; however the exogenous approach was taken to simplify implementation of both the simulation and learning algorithm.

4.1.1 Software Architecture

This section describes the technical details of how the conceptual architecture was implemented as a software system. The agent system is implemented using a 3D game engine. The system runs as two independent asynchronous software processes, one for the agent and one for the simulation itself. Figure 4.2 shows the main software components and how they interact. In the diagram software components are shown as blocks (sub-components are shown as blocks within parent components). Implementation languages are shown in brackets. A block on top of another indicates a runtime dependency. Arrows represent the flow of data (observations and actions) between the two processes.

The agent process is implemented using the Scala programming language and runs on the open-source Scala runtime[2]. The skill and vision system are not located in the agent process as might be expected because it is more convenient to implement them within the simulation where they have direct access to the data structures used by the simulation.

[2]Scala is a hybrid object-functional language implemented on the Java Virtual Machine. The Scala runtime consists of a Java virtual machine and Scala libraries. See www.scala-lang.org for details.

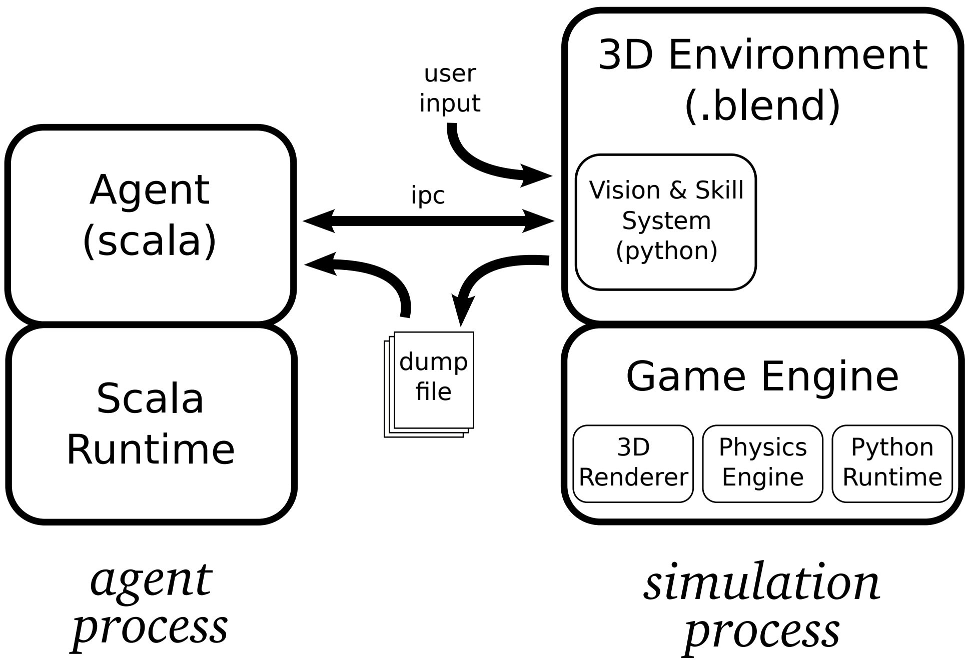


Figure 4.2: Software Architecture

The simulation process is implemented using the Blender Game Engine program.[3] The game engine executes a custom designed environment. The environment is a data file (or 'blend') created in a 3D graphics package. It contains information about a particular scene in which the agent can interact (a kitchen for example). The data file specifies what objects are in the scene, their properties and how they interact with the agent and with each other.

Embedded within the environment is custom code for observing and interacting with objects, it implements the agent's vision and skill systems. These sub-systems are written in the Python programming language using a game engine API for accessing object data and events (such as object position and object collisions). The Python code is executed by a Python

[3]Blender is an open-source 3D graphics and game engine software package. See www.blender.org for details.

runtime embedded within the game engine.[4]

The game engine simulates an environment in real time. Objects are constantly modified according to the rules of environment. A physics engine ('Bullet Physics'[5]) is integrated into the game engine and is used to calculate rigid body physical interactions. The physics engine simulates various physical phenomena including gravity, friction, inertia and restitution.

An environment can be visualized in real-time using a 3-dimensional renderer also embedded within the game engine. The renderer allows a human teacher to observe and interact with the scene in real time (as though playing a computer game).

The agent and simulation use inter-process communication ('ipc') to send and receive observation and action information. The communication is implemented using network sockets in which the game engine acts as a server and the agent as its client. At any time the agent can asynchronously send an action to the environment and it will be executed at the next available opportunity. In response, the game engine is constantly generating and sending snapshots of world observations, taken at regular intervals, back to the agent. This communication mechanism makes it possible for the agent to implement an interactive and online learning algorithm. The simulation can also receive action instructions from a user via keyboard and mouse commands. The actions are executed as though the agent had sent the action request.

The vision system can optionally output serialized observations to a 'dump file' as an alternative to real time communication with the agent. In this case the game engine continues to construct regular snapshots but stores them directly in a file rather than sending them to the agent. The agent can then be made to process the observations at a later time. Obviously, using the dump file prevents the agent from interacting with the

[4]Python is a general purpose scripting language. See www.python.org for details.

[5]Bullet is an open-source physics library. See www.bulletphysics.org for details.

simulation while it is learning. However, the dump files have several practical advantages: they allow sequences of snapshots to be run multiple times against different learning parameters in order to compare results; they can run significantly faster than the real time simulation; and finally, they avoid the problem of the simulation crashing (which occasionally occurs on longer runs) while the agent is learning.

4.2 Simulation

Simulated environments are executed using the Blender game engine. The game engine uses a combination of custom code and the built in physics engine to simulate world dynamics. Each environment is constructed by hand using a 3-dimensional graphics package. Constructing an environment involves specifying the objects contained within it, including their shape and physical properties (e.g. weight), along with code that specifies any custom behaviour or interactivity. It is possible to construct many different environments, for example, figure 4.3 shows an everyday kitchen environment with various working devices. There are also a number of pre-constructed environments available in the form of games that could be modified to work with the skill and vision system.

The game engine executes an environment by recalculating the state of the world at regular fixed length intervals. The time between updates, or 'simulation interval', determines the precision of the physics simulation. Decreasing the interval improves precision but reduces the available cpu time for competing threads which impacts on interactivity and rendering frame rate.

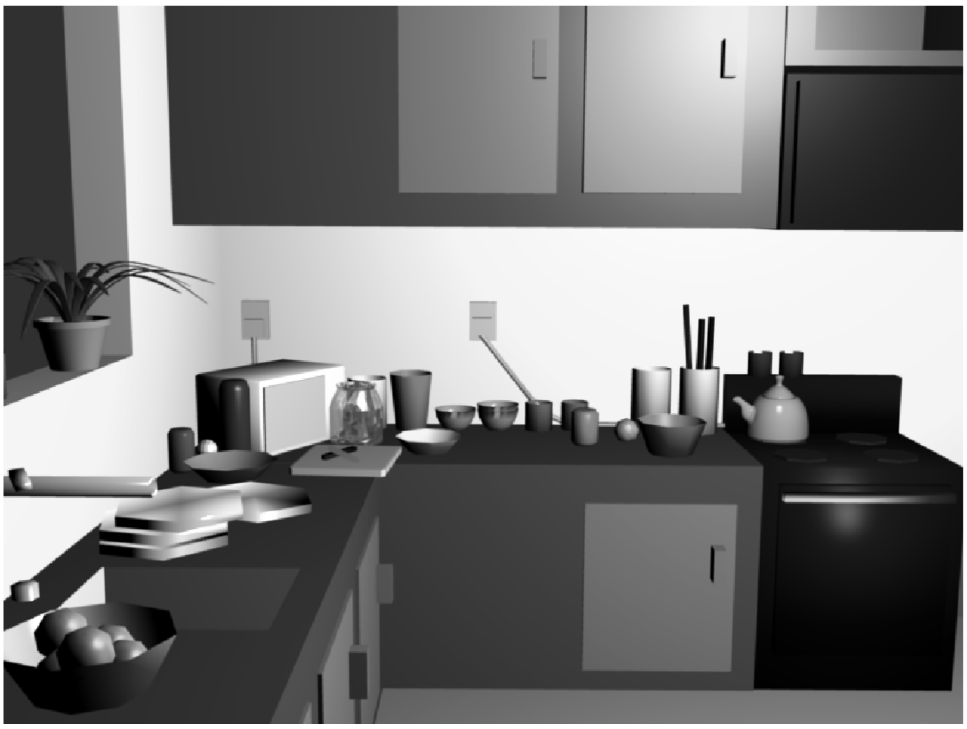
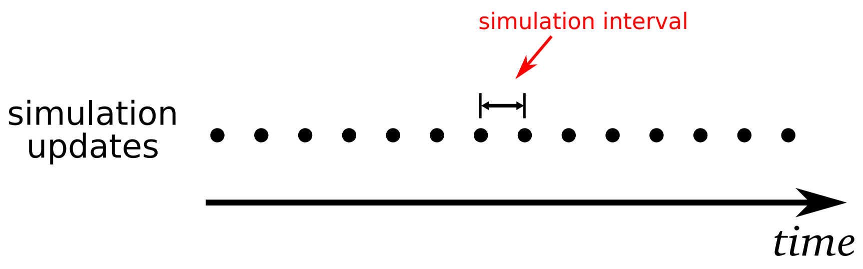


Figure 4.3: A Simulated Kitchen Environment



A new world state is calculated by updating each object in the environment. The physical properties of each object are updated according to its previous state and any collisions that may have occurred since the previous state. Dynamic physical properties include: location, rotation, velocity and torque. The simulation also updates any custom properties that have been specified as part of the object's state. The simulation state can also be updated externally via the skill system.

4.2.1 Rigid Bodies

The physics engine implements a simulation of rigid body physics. The shape of each rigid body object is specified as 3-dimensional triangle mesh. The surface of the mesh is used as the physical bounds of the object for calculating collisions with other objects. The following describe how physical phenomena are simulated by the physics engine:

Inertia is simulated by maintaining an object's momentum in the absence of any force acting on it. The physics engine allows objects to specify linear and rotational dampening, which can be used to approximate air drag or enforce a maximum velocity.

Dry Friction is simulated by applying a resisting force on objects as they slide against each other. Objects have zero friction by default but may be flagged as causing friction and given a custom relative coefficient. This allows similar shaped objects of varying materials to behave differently during a simulation, e.g. objects can be pushed across a polished floor more easily than across a carpeted floor. Other types of friction (e.g. lubricated friction) are not simulated but can be roughly approximated by reducing an object's dry friction coefficient appropriately.

Restitution is simulated by applying force to an object subsequent to a collision. Each object has a custom value representing its 'bounciness'. Restitution allows objects such as bouncing balls to be simulated.

Gravity acts on objects pulling them in the direction of the negative z-axis (i.e. the environment's center of gravity is at $-\infty$). Gravity can be reduced to simulate 'weightless' environments.

Collisions are detected by calculating surface intersections. Object collision meshes may be concave or convex. Objects can be flagged as 'static' in which case they can collide with other objects but are fixed in position.

Impulses simulate a directional force being applied to an object. A linear or rotational impulse may be applied to an object at any time. An object's mass property is used to calculate the object's resulting velocity and torque. Impulses can be used to represent an agent 'pushing' an object with its manipulator.

Linkages allow objects to be joined with specified degrees of freedom. Objects can be joined with ball, hinge and generic (up to 6 degrees of freedom) linkages. The linkage restricts the movement of the object relative to the other. Linkages allows realistic modelling of mechanisms such as cupboard doors.

The physics engine supports soft-body simulation to a limited extent, but at the time of writing it was too unstable to be usefully used in this project.

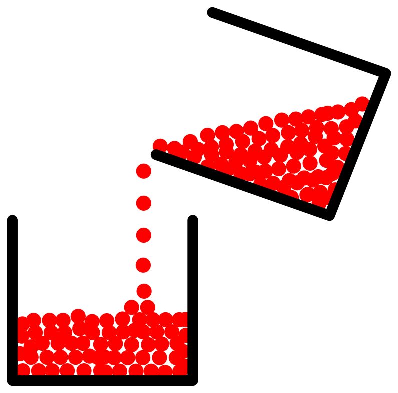


Figure 4.4: Approximating Liquid Flow with Particles

4.2.2 Liquids

Liquids occur in many everyday environments and are important aspect of naive physics. Unfortunately the Bullet physics engine does not support *real-time* simulation of liquids because algorithms that provide visually attractive results are too computationally intensive. Fortunately it is possible to crudely approximate liquids using a particle system and rigid body physics. The resulting simulation is not visually attractive, but the behaviour is sufficiently realistic for the purposes of extracting qualitative observations.

Liquids are simulated as a group of rigid body spherical objects. The spheres move freely and flow in a similar manner to a liquid. The 'liquid' 'pools' in containers and will 'flow' from one container to another when 'poured'. Figure 4.4 illustrates some liquid in the process of being transferred from one container to another. The size of the spheres determines the quality of the simulation, smaller spheres are more realistic but require more computation to execute the simulation.

Collision detection can be used to allow other objects to 'absorb' a liquid. The absorbing object is flagged as 'wet' or 'saturated' and the colliding sphere is removed from the simulation as it is consumed.

Liquid sources are modelled by making a special object which generates spheres at a specified rate. This allows the modelling of objects such as taps. Similarly, special sphere consuming objects can be created to model drains.

4.2.3 Devices

Devices are objects that have some state that is updated using custom logic written specifically for the particular environment. A example of a simple device is a light bulb, it has two states: lit or unlit. The simulation executes custom logic to update the state of the bulb, for example, a connected switch may be flicked causing the simulation to toggle the state of the bulb.

A group of devices can be combined to create more complex composite objects, for example an electric oven can be modelled as a variety of switches and heating elements. The state of the oven at any given time is determined by the custom logic controlling the devices.

Device objects are also physical objects and therefore interact with the physics engine as normal.

4.3 Vision System

The vision system is designed to fulfill the requirements outlined at the beginning of this chapter, it is high-level, object based, plausible and assumes an open-world. The vision system is built using the game engine's api, which gives access to low-level object information. The object information is transformed by the vision system into high-level abstract 'observables', for example, a set of 3-dimensional co-ordinates describing an object's surface mesh can be used to calculate if two objects are 'touching' one another or not. The agent observes that the objects are touching but is oblivious to the co-ordinate information used to construct the observation.

A set of observations made at a particular moment in time is called an

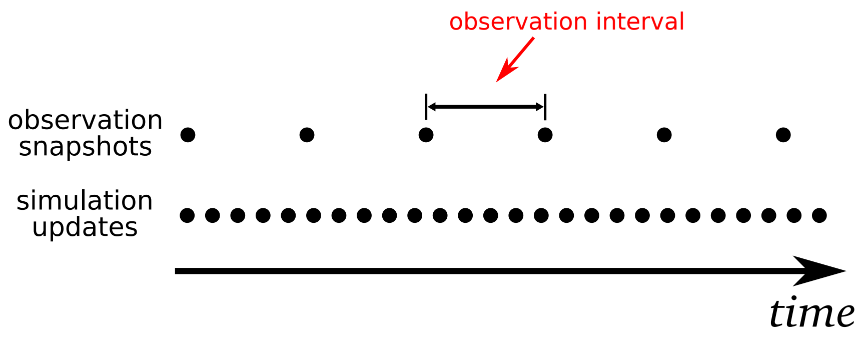


Figure 4.5: Observation Interval

observation snapshot. The vision system provides a series of observation snapshots made at regular intervals. Figure 4.5 shows the observation interval in relation to the simulation interval. The observation interval is significantly longer than the simulation interval, this allows the world state to change many times a second, whereas snapshots may be observed only once or twice a second.

The temporal resolution of the vision system presents a trade-off between computation time and observation continuity (qualitative states can be missed if there are multiple transitions between adjacent observation snapshots). Missing occasional transitions is acceptable because it is an expected feature of a *plausible* vision system used on a physical agent (especially because multiple transitions can occur instantaneously, for example, when a measure passes through a landmark value). Missing small numbers of transitions is not problematic for learning: some missing transitions can be inferred from the adjacent snapshots or they can be found by re-observing the process in question. A long enough interval will degrade learning significantly but this would have to be an unrealistically long gap between snapshots. Figure 4.6 illustrates a missed qualitative state due to multiple transitions between snapshots.

Observation snapshots generated by the vision system represent only a partial state of the world. Objects may be hidden from view (this is implemented by flagging objects that cannot be seen). Additionally, objects

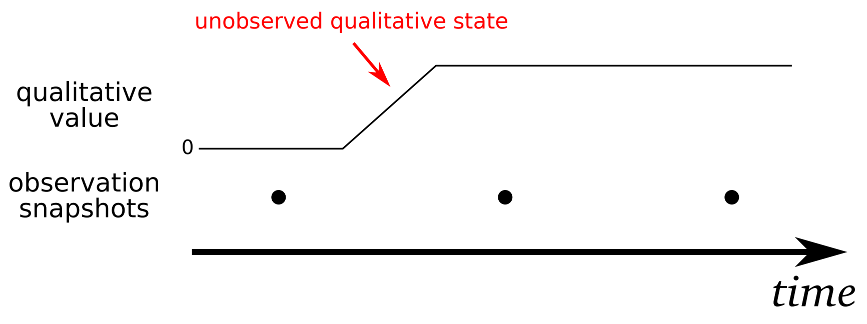


Figure 4.6: Multiple Transitions Between Snapshots

may have properties that are flagged as ‘unobservable’ which the vision system simply ignores.

An important aspect of the design of the vision system is deciding what is observable in a given environment. Ideally the agent would learn what are the useful features of the environment that should be observed. This is a difficult task and is therefore outside the scope of this thesis which focuses on learning the behaviour of the world *once* suitable observable features have been identified. Given the conceptual architecture of the learning system there is no objective way to choose what properties and relationships of objects should be observable. Therefore, observables are chosen arbitrarily for each environment using rules of thumb which are intended to help avoid artificially simplifying the learning problem. Simplification of the learning problem is the main concern when choosing rather than learning the environment’s observable features. Observables are chosen using the following rules of thumb:

- Include a variety of observables that are irrelevant to the behaviour being learned. Obviously the task of identifying relevant properties and relationships is much simpler if all observables are relevant.

- Avoid unrealistic or implausible observables. The chosen observables should reflect the limitations of a plausible robotic vision system. For example the ability to directly observe the coefficient of friction of a material is not realistic, whereas the ability to observe the material's colour is.

- Observe qualitative observables in terms of 'natural' quantity spaces. Qualitative variables are specified in terms of a quantity space which describes the possible values that the variable can take. Choosing natural landmarks as qualitative comparison points avoids choosing arbitrary comparison points which may be tailored to a specific behaviour and therefore simplify the learning task. For example, 'above', 'level' and 'below' is a natural way of measuring relative position because the influence of gravity. These types of comparison points are chosen in preference to values such as '20cm to the right of' or '10cm behind' which have no obvious justification.

4.3.1 Implementing Observables to Support the Q-System Representation

The types of observables produced by the vision system are specified by the Q-System representation. Figure 4.7 shows the hierarchy of types used in the Q-System. The two main branches 'fluents' and 'events' are distinguished by their temporal behaviour with respect to the vision system's snapshots. Fluents represent state as it currently exists at the time of the snapshot. Fluents will change state instantaneously between snapshots; they cannot be observed between states. Events are the counterpart to fluents: they occur between snapshots and are not part of the current world state; instead they are transient and observed to have occurred *between* the previous and current snapshots.

The following list describes how each observation type is implemented in the vision system:

Existence observations represent the existence of an object in the agent's field of view. Existence observations have no value, they are present

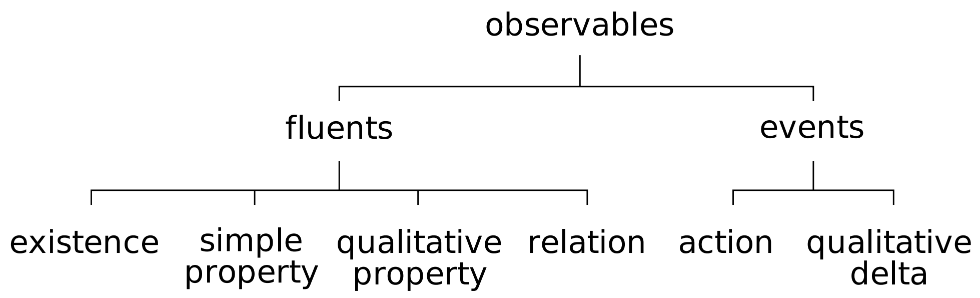


Figure 4.7: Hierarchy of Observables

in the observation snapshot or omitted if the object is hidden or destroyed. Each object in the simulation is assigned an identifier which is used by the vision system to track objects between snapshots. This follows from a principle of the representation that individual objects can be tracked across time. Object 'trackers' Kuipers et al. [29] are an example of how this ability could be implemented in a real vision system.

Properties are discrete valued variables describing some aspect of an object. Property values are calculated by the vision system and inserted into observation snapshots. An example property is 'colour' which takes a value from a fixed set of colours.

Relations are discrete variables describing some aspect of how two objects relate to each other. The vision system calculates values for all pairs of observable objects. An example relation is 'touching' which takes the values 'yes' or 'no'.

Qualitative Properties are discrete variables that have two components: a qualitative value from a quantity space and a qualitative derivative describing the underlying variable's qualitative rate of change (increasing, decreasing or steady). The vision system calculates the qualitative value using the underlying real value, which is normally available from the simulation (an object's velocity for example). Figure 4.8 illustrates how changes in a simulated real valued variable are observed qualitatively over time. The variable starts at 0 and after a pause increases before finally leveling off. Three observation snapshots are shown along with the resulting qualitative observations.

Qualitative derivatives are calculated by either recording the underlying value from the previous simulation update and then comparing with the current value; or alternatively the value can be inferred from known influences (for example, an isolated object travelling over a surface can be assumed to be decreasing in speed due to friction).

Qualitative Deltas are events that represent how a qualitative property has changed between snapshots (cf. a qualitative *derivative* which specifies how a qualitative property is *currently* changing). The vision system calculates qualitative delta observations by recording the value of each qualitative property at each snapshot and comparing with the previous snapshot's value. The delta is assigned a value of increased, decreased or unchanged.

Action Events are events that represent an action having occurred between snapshots. The vision system reports any action executions including the target object(s). If multiple actions occur during an observation interval then they are all included in the observation snapshot, but no information is observed regarding the order in which they occurred. Some actions take longer than a single observation interval to execute in which case the vision system is paused until the action completes, thus fulfilling the assumption of atomic action execution.

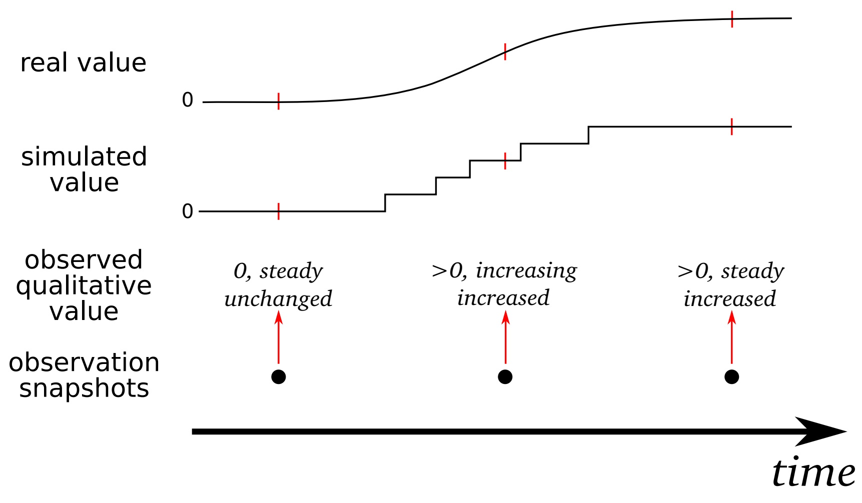


Figure 4.8: Qualitative Observations of a Simulated Variable

4.3.2 Observing Qualitative Shape

Qualitative shape is an important aspect of interpreting everyday environments. For example, we can deduce that a tennis ball will roll much better than a brick because of their respective shapes. The vision system implements a selection of qualitative shape observations which describe the objects in the simulation. The purpose of the observations is to allow some 'interesting' behaviour in the simulation to be observed qualitatively.

There are many ways to model shape qualitatively and there is no widely accepted general purpose representation, therefore a novel set of shape observations has been implemented in the vision system.[6] Because there is no requirement for the observations to be rich enough to describe all nuances of dynamic behaviour [7], the shape observations are incomplete in this sense. Similarly, there is no requirement for the shape representation to be particularly efficient (either in terms of computational require-

[6]Cohn [8] provides a survey of the many approaches to modelling qualitative space and shape.

[7]In fact the poverty conjecture (see section 2.3.2) suggests that is a not even a possibility to describe all behaviour with purely qualitative information.

ments or in terms of representational conciseness and redundancy) so long as it is practical. Despite these relaxed requirements, the chosen observations must be sufficient for the agent to learn a 'naive' behavioural model of object interactions; therefore, the types of shape observations have been arbitrarily selected with this goal in mind.

The observable qualitative shape properties are arranged in a hierarchy in which each shape class inherits shape properties from those above and implements some new properties of its own. Figure 4.9 shows the shape hierarchy implemented in the vision system. Each object in the simulation is assigned to a shape class which determines the observations that may be made about the object's shape. The following shape classes are implemented in the vision system:

Shape objects have two observables: 'centroid-supported' and 'centroid-vertically-aligned'. Every object is assigned a 'centroid' which roughly locates the center of the volume of space occupied by the object (e.g. the center of a sphere). Centroid-supported is a property and is true whenever the surface directly beneath the centroid is touching another object. Centroid-vertically-aligned is a relation that is true whenever two objects' centroids are directly aligned along the Z axis. All other classes inherit from the Shape class.

Convex-Regular objects are convex and have no curves.[8] They have 2 properties: 'flat-top' and 'flat-bottom'. Flat-top is true if the highest face of the object is orthogonal to the Z axis; flat-bottom is true if the lowest face of the object is orthogonal to the Z axis. Convex-Regular objects inherit from the Shape class.

Cuboid objects have six orthogonal flat faces and have the property 'orientation' which can take the values 'big-face-up', 'middle-face-up',

[8]Technically none of the simulated objects have curves because they are all constructed using triangle meshes, however, the shape hierarchy uses 'curve-like triangle mesh' as an abstraction of truly curved.

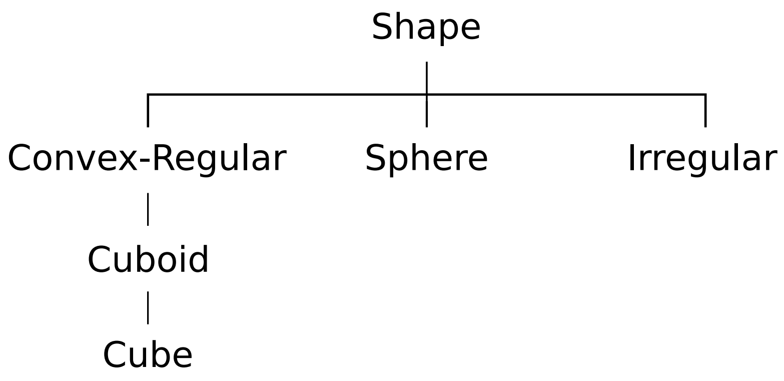


Figure 4.9: Hierarchy of Shape Classes

'small-face-up'. The property is determined by finding the face which is closest to orthogonal with the Z axis. The 'size' of a face corresponds to its surface area, so a cuboid shaped book is small-face-up when stacked on a book shelf and big-face-up when laying flat on a desk. Cuboids inherit from the Convex-Regular class.

Cube objects have six equal sized orthogonal flat faces. The orientation value of a cube is always large-face-up. Cube objects inherit from the Cuboid class.

Sphere objects are (approximately) spherical. Spheres inherit from the Shape class.

Irregular objects all objects which do not fit the requirements of any other class. Irregular objects inherit from the Shape class.

The object shape properties are calculated using the object location and orientation information provided by the simulation api.

4.3.3 Observing Qualitative Liquids

Liquids can be qualitatively observed in terms of flow rates, contained amounts and height levels. Observations about liquids are assigned to the objects generating and containing them rather than creating a 'liquid object' instance and assigning properties to it. This avoids the difficult problem of determining the existence and extent of liquid objects. For example, is the water in a pipe an object, is it 'connected' to the water flowing from the pipe, or are they the same object? [9]

Flow rates are observed as qualitative variables with a quantity space of $[0,\infty]$ (i.e. the value can be 0 or >0). Flow rates are observed by inspecting the water sphere generator within the simulation. The flow rate property is assigned to the generating object (a tap for example).

Container objects such as basins have two qualitatively proportional properties describing the liquid inside them. The 'amount-of-liquid' and 'liquid-height' properties represent the amount of liquid in the container and how high the water has risen from the bottom. They both use the quantity space [0,MAX] (giving possible values 0, 0-to-MAX, and MAX). The values are calculated by counting the liquid spheres known to be within the confines of the container object. Containers can overflow when they are full, leading to liquid spheres escaping the container and collecting elsewhere. The liquid observables are not affected by overflows (i.e. the amount-of-liquid and liquid-height continue to be observed as MAX while a container is overflowing).

Calculating the qualitative derivative of the liquid-height and amount-of-liquid must be done by inferring the instantaneous change from the known influences (i.e. any inflows or drains). This is because a *continuous* influence may be generating or consuming discrete spheres at a rate of less than one per *simulation* update, thus making the usual comparison of adjacent simulation states inaccurate (e.g. no sphere appeared between updates indicating the water level is steady when it should be rising). Figure 4.10 illustrates the problem of incorrectly calculating a qualitative derivative of liquid height using adjacent simulation updates.

[9]See Davis[9] for a detailed investigation of these issues.

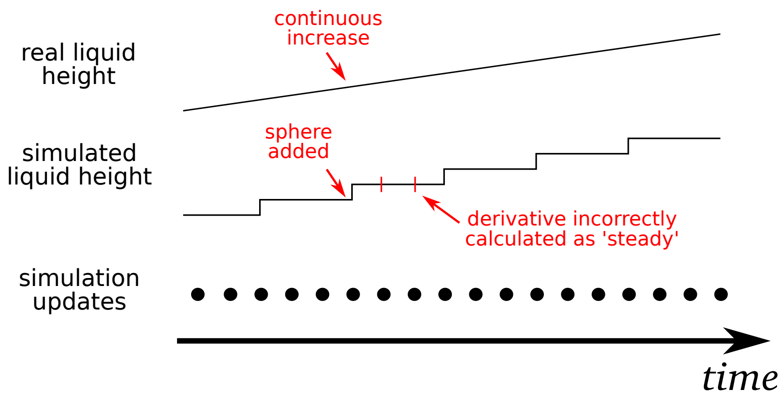


Figure 4.10: Incorrect Calculation of Liquid-Height Derivative

4.3.4 Observing Qualitative Kinetics

The qualitative movement of objects is observed by the vision system using two directional properties, vertical-movement and horizontal-movement, and a single object-speed property. The vertical-movement property takes the values UP, DOWN and NONE. The horizontal-movement property takes the values YES or NO, indicating whether it is moving in the XY plane. Together these properties give an object six possible absolute directions in which to travel.

The object-speed property is a qualitative variable which takes values of either 0 or >0. The speed observation is calculated directly from the simulation's object velocity attribute. The qualitative derivative (i.e. acceleration) is calculated by comparing speed at adjacent simulation updates to provide an 'instantaneous' (at least in the agent's mind) rate of change.

Relative movement can be observed using relationships between objects. For example, the relationship distance-between-objects takes the values increasing, decreasing or not changing. Also, the relationship on-collision-course can be used to indicate when two objects are expected to collide if neither of them change speed or direction.

4.4 Skill System

The skill system implements a high level action interface that allows an agent (or user) to request atomic action executions on target objects and have them realised in the simulation using low level api calls. The requirements of the skill system are that it executes actions atomically (i.e. actions are executed between observation snapshots and never across them), that it executes 'high level' actions which are assumed to have been previously learned (i.e. the agent can request a 'pick up object X' action and not be concerned with the underlying fine level control of arm and gripper actuators), and that the outcome of actions is non-deterministic (for example, if the agent pushes a stack of objects, sometimes they fall over and sometimes they do not). The agent is assumed to have a skillful manipulator arm that can interact with objects in a similar manner to a human.

The skill system is implemented as an independent thread within the simulation process, this allows it to execute actions asynchronously with respect to simulation updates and observation snapshots. Semaphores are used to communicate with the observer thread, allowing the timing of observation snapshots and action executions to be co-ordinated when required. The skill system acts as a listener and waits for action execution requests. The requests can come from either the agent process or directly from user input. The actual execution of an action and its subsequent observation is identical whether the requests was generated by the agent or a user. This enables the user to guide the agent by effectively choosing actions for it. The skill system implements action execution requests by using the simulation's api to move, apply force to, or update the internal state of objects.

Actions may be executed at any time during the simulation but they are prevented from occurring simultaneously, the agent cannot multi-task. Some actions may be implemented as composite actions, for example the put-on action is implemented as a sequence of pick-up, move and put-down actions. This composition is useful for implementation purposes but it is not visible to the agent. The agent is not aware that a put-on action is implemented as a decomposition, however there is nothing preventing the agent from learning and implementing its own action compositions if required.

To ensure atomicity actions are always completed before the next observation snapshot. This normally requires the observation snapshot to be delayed because actions typically are longer than the observation interval. The delay has does not need to be explicitly observed by the agent because it assumes a similar but *non-fixed* interval between states.

Non-deterministic outcomes are implemented by two mechanisms, firstly by ensuring certain actions are executed slightly differently every time, and secondly by relying on the non-determinism of object interactions in the underlying simulation. The 'move' action is an example of an action that is executed slightly differently each time because the object is moved to the approximate target location rather than the precise location. This can lead to different outcomes depending on the other objects in the vicinity of the target location.

Occasionally invalid actions will be requested. This can happen if a target object has ceased to exist in the time between choosing an action and its execution. In these cases no action is performed and no action is observed. Invalid actions are distinct from actions which 'fail' in some way, such failing actions are observed normally and not explicitly flagged as failed. For example, a pick-up action may fail when it drops the target object during the action execution; in this case the agent will observe a 'successful' action with an unusual outcome.

Multiple actions are allowed to occur between snapshots but this rarely happens because most actions are longer than the observation interval. If the situation does arise then the all actions are observed in the following snapshot as having occurred 'simultaneously' but they are not ordered. The agent is unable to observe the order of events at a resolution greater

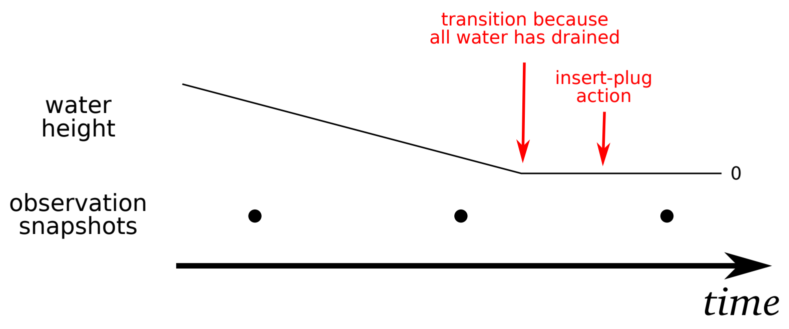


Figure 4.11: Simultaneous Observation of Dynamic Transition and Action

than the observation interval. This lack of temporal resolution results in misleading observations when action executions occur coincidentally at the same time as qualitative state transitions generated from the simulation dynamics. There is insufficient information in the snapshots to enable the observer to disambiguate the causal effects of the action and the ongoing effects of the underlying world dynamics (this is a similar problem to the problem of missed qualitative states discussed in section 4.3). Figure 4.11 illustrates an example of this problem with respect to action timing. The scenario involves a sink that is emptying of water through a drain. The insert-plug action is executed coincidentally during the same observation interval as the last of the water drains from the sink. The resulting observation snapshot indicates that inserting the plug *caused* the sink to become empty. These misleading observations must be resolved by the agent if it is to learn useful models.

4.4.1 Example Action Implementation

This section describes a typical action execution in detail to show how the simulation api is used to execute the action. The example action is the put-on action which has two target objects, the object to move 'X', and the object on which it will be placed 'Y'. The following steps are carried out to execute the action:

1. The observation thread is paused.

2. Object X is grabbed by the agent. This involves changing X from a dynamic object which responds to physical forces that act upon it, to a static object which collides with and moves other objects but cannot be moved itself by the physics engine. Making the object static simulates the effect of the agent holding the block securely. (The physics engine is suspended while the object is changed, this is to avoid unrealistic interactions between objects when X is suddenly replaced by a new static version with different physical properties.)

3. Object X is picked up. This involves moving the static object at a constant rate to until a specified height is reached. Any objects on top of X are picked up with it. A timer is started to periodically check if the object is in the approximate location. The use of the timer introduces some imprecision in the action because there is some variability in the delay between position checks.

4. The centroid of object X is aligned above object Y. Again a timer is used to check when the object is in position. During this process the object may collide with other objects, also any objects on top of X may fall off due to the effects of momentum (and also dependent on the amount of friction between the objects).

5. Object X is lowered until it is close to touching an object. It cannot be lowered all the way because X is still a static object and this would be equivalent to the agent moving X down with *zero* give, causing the underlying object to be forced away (explosively due to limitations in the physics engine).

6. Object X is dropped. X is changed back from a static object to a dynamic object. Gravity takes effect and X falls the remaining short distance to land (hopefully) on top of Y.

7. The observation thread is restarted. The put-on action event is added to the current observation snapshot.

It can be seen from the example that the actual outcome of the put-on action is dependent on a variety of complex interactions between control commands, other objects, and the physics engine dynamics.

4.5 Test Environments

Two detailed test environments have been created for the purpose of developing and testing learning algorithms based on the Q-System representation. The environments are a kitchen world containing various devices found in everyday kitchens and a toy world containing various differently shaped toy blocks. These environments were chosen because they exhibit quite different sorts of behaviours, the kitchen world contains controllable devices and liquids, whereas the toy world contains objects which interact in complex ways.

The agent can interact with and observe the objects in the environments using the vision and skill systems previously described. The environments are created using a 3-dimensional graphics package and incorporate custom code that implements the behaviour of devices. The game engine executes the environments as described in section 4.1.1.

The environments have been designed to simulate the various types of observable behaviours and interactions that make learning in realistic worlds difficult. Some of the features can be turned on or off depending on the particular learning scenario. This allows each environment to exhibit a variable level of learning difficulty. For example the weight of objects, which is normally a hidden variable, can be made observable if required.

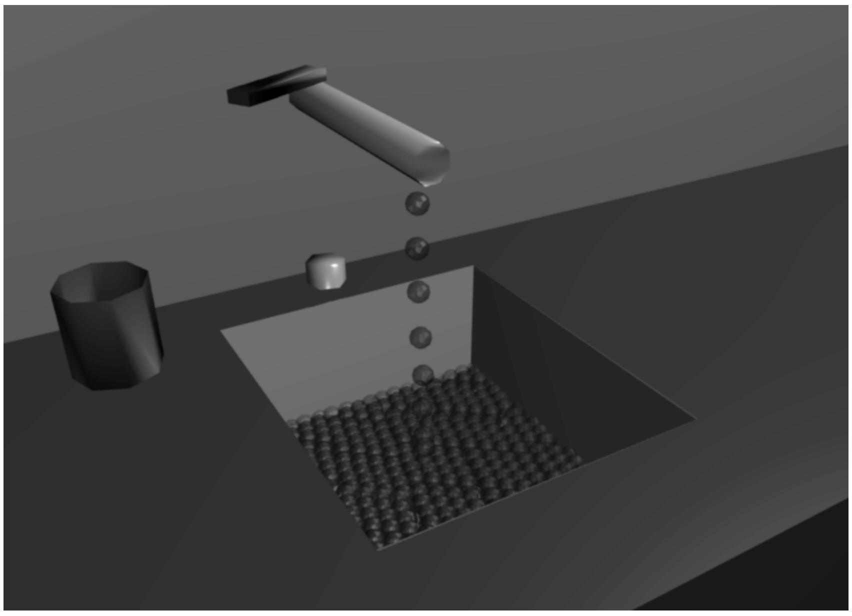


Figure 4.12: Kitchen Environment Tap and Sink System

4.5.1 Kitchen World

The kitchen world has been designed to allow an agent to explore a typical kitchen environment, and in particular the various types of stateful 'systems' that can be found in them. Many kitchen devices are stateful, for example an oven has a variety of controls that determine which elements are heating.

The environment contains a tap and sink system with a working simulation of water flow and water containment (implemented using the sphere approximation technique, see section 4.2.2 for details). The agent or teacher can turn the tap clockwise or anti-clockwise to affect the rate of flow. A plug can be inserted or removed allowing water to drain. The sink can overflow and spill water onto the surrounding bench and floor. Other containers such as a cup can be placed beneath the flow to allow them to be also filled with liquid. The tap and sink system is shown in figure 4.12.

Objects in the kitchen world can be moved around by the agent from location to location. For example, a pan can placed on a hob on the oven top. Also, the kitchen world can be reconfigured to give the agent different learning scenarios in the same environment. For example, the type of tap can be changed from one with a single control to one with multiple controls. This gives the agent the opportunity to generalize similar but different systems.

4.5.2 Toy World

The toy world has been designed to allow an agent to explore rigid body kinetics in terms of qualitative space and shape. The world contains a variety of objects including blocks, planks, balls, cones, hoops as well as more irregular objects. The toys are all observed using qualitative shape descriptions described in section 4.3.2. The behaviour of the objects is simulated very realistically by the physics engine which results in subtle and complex observable behaviour. Figure 4.13 is a screenshot of the toy world simulation.

Several manipulative actions are available to the agent in the toy world. These include: grabbing and releasing objects; picking up objects and dropping them; stacking objects on another; pushing objects laterally (either away from the agent or towards another object); throwing objects (again, either away from the agent or towards others). Toy world supports learning about various scenarios. There are many possibilities but typical examples range from exploring the effects of momentum and collisions on moving objects, to building complex structures from stacks of blocks.

The environment is similar to the long standing artificial intelligence test environment known as 'blocks world'. Blocks world has been used as a learning environment since the inception of the research field, classic examples include Winograd's SHRDLU [42] and Winston's arch learner [43]. The toy world environment differs from these classic environments

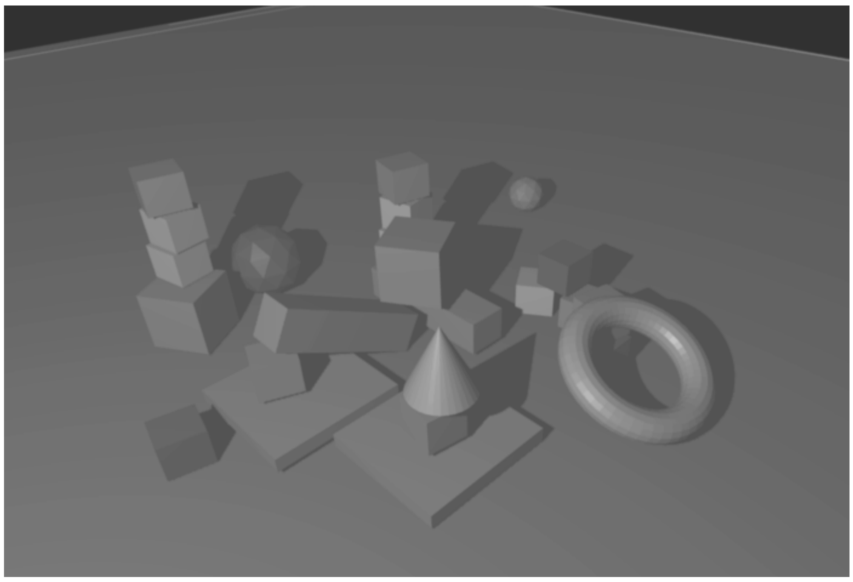


Figure 4.13: Interactive Toy World Simulation

in several important ways: it supports an effectively unlimited numbers of objects, imprecise actions, hidden state, stochastic effects, unlimited object shapes, and finally, much richer/finer object simulation. The later is particularly significant because the required computational power and real-time physics simulation software has been unavailable to the majority of researchers until relatively recently. Freely available software and modern graphics/physics co-processors has enabled more interesting and difficult problems like this to be approached.

Chapter 5

Histories

Recording of histories is a key step in learning generalised Q-Systems. This chapter describes what histories are and how they are used by the learning system. It also describes how histories are extracted from the simulation and describes some examples in detail. Finally the chapter describes the heuristics used by history recorder and how they affect the types of histories that are extracted.

5.1 What is a History

Histories are snippets of world behaviour as observed by an agent. They are temporally and contextually restricted sets of observed state transitions. The temporal restriction limits a history to a certain time frame during which a limited number of transitions occur. Without this restriction, histories would grow endlessly as the agent continues adding transitions as it observes the world. The temporal restriction is used to delimit the start and end of an 'interesting' or useful (with respect to the learning algorithm) sequence of behaviour.

The contextual restriction limits a history to a group of specific objects. Each state in the history contains assertions about objects in the context only, assertions concerning other observable objects are removed from the history. The contextual restriction allows a history to be focused on a small group of objects and their behaviour, thus significantly reducing the number of observations involved. This is necessary because the agent can observe an effectively unlimited number of features in the world, which means working with complete world states is not practical.

The purpose of recording histories is to generate specific examples of world behaviour which can be generalised into useful Q-System models. The transitions in a history are identical to those used in the Q-Systems representation (they are annotated with the actions, or absence of actions, that occurred between states). This means a history *is* a simple Q-System which describes the behaviour of some specific objects. The Q-System context is simply the set of assertions which do not change in any of the history's states.

A set of histories is used as a dataset for the Q-System generalizer. Typically the generalizer will process histories by replacing specific objects with variables, dropping irrelevant assertions from the context, and combining states from several histories to produce a more complete picture of system behaviour.

Datasets of Histories vs Datasets of Individual Transitions

Typically, existing action learning systems use sets of individual transitions as datasets (where a transition is a pre-state, an action, and a post-state). These transitions can be thought of as histories which are temporally restricted to a single time step but contextually unrestricted (i.e. states are descriptions of the entire world). Such learning systems assume that the observable world is restricted to relevant objects only or that complete world descriptions are small enough to be processed efficiently. In the context of learning in complex realistic worlds, these assumptions ignore a vital component of a complete learning system and are unrealistic.

Including a history generation step with contextual restriction is one way to address the problem of determining which objects in the world should be included in generalised models and avoids the problem of processing complete world descriptions.

5.2 Example Histories

Histories are generated from the simulation by recording sequences of transitions. The agent connects to the simulation, observes actions being executed (either by itself or an instructor) and builds histories in real time. At each discrete time step the agent can start a new history, update any previously started histories, or stop any previously started histories. This section describes some example histories generated from the simulation.

Rolling Ball Example

This example illustrates how the agent observes the simple behaviour of an object as a history. The scenario involves a simulated object responding to the actions of the agent. The history is generated from the simulation which executes a detailed realistic approximation of a ball rolling over a surface after having some lateral force applied (i.e. it was pushed by the agent). The agent records the state of the ball at each observation snapshot provided by the simulated vision system. In this instance the history's context is restricted to just the ball and the time-frame is restricted to just before and just after the ball was moving. The resulting system is shown in figure 5.1. Table 5.1 shows the qualitative deltas (see section 3.2.3) associated with the transitions in the rolling ball system (note that '...' is used to indicate the 'time passes' transition, i.e. the absence of an action).

The rolling ball system has three states. The ball is stationary to begin with (state **a**). It is then pushed by the agent which causes it to start rolling (state **c**). After some time, the ball slows to a point where it is just barely

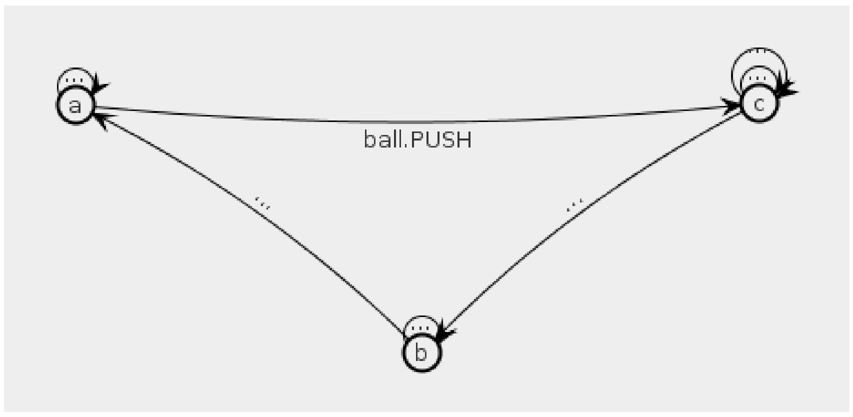


Figure 5.1: Behaviour Graph of System Generated from Observing and Interacting with a Rolling Ball. The system has three states: **a**. the ball is stationary; **b**. the ball is moving but its effective speed is zero; **c**. the ball is moving with non-zero speed (see text for explanation).

transition			**deltas**	
a →	...	→ a		
a →	ball.PUSH	→ c	ball.speed	+
b →	...	→ a		
b →	...	→ b		
c →	...	→ b		
c →	...	→ c		
c →	...	→ c	ball.speed	-

Table 5.1: Transitions from Rolling Ball History

moving, at this point the vision system has trouble discerning if the ball is actually stopped and interprets this as the ball having some horizontal movement but effectively 0 speed[1], state **b**. Eventually the ball is quiescent and returns to its original state.

The system has several *time-passes-and-nothing-(qualitative)-changes* transitions. These are represented in the diagram as self-loops with no action. Such transitions are included in the system to distinguish between transient and persistent states. All the states in the ball system are persistent indicating that they were observed in at least two consecutive observation snapshots. State **c** has two such transitions. It can be seen from the transition table that these are distinguished by the qualitative deltas observed coinciding with them. One of the self-loops was observed with a qualitative delta indicating that the speed of ball slowed between snapshots. The other self-loop has no qualitative delta associated with it, this is because the ball was moving too slowly for a change in speed to be observed.

The rolling ball system highlights the subtlety of interaction between the vision system and simulation. Together they have generated a sequence of observations that has led to an unexpected state (**b**) and an unexpected transition (the self-loop on **c** with no delta) which would not be included in a 'clean' or idealised model of the rolling ball system. A clean model, the kind that might be expected to be observed, would include only two states **a** and **c**.

This 'unexpected' state shows that the simulated vision system has some realistic properties. A real (artifical or human) vision system would make the same kinds of mistakes. For example, humans have limited precision when recognizing movement and sometimes will not notice that a moving object is actually moving (albeit very slowly). Eventually, the fact that the object did move might be inferred by some other means (by

[1]The speed of the ball is not actually zero but it is not significant enough for the vision system to report it as non-zero. Note that speed and horizontal movement are calculated independently and therefore state **b** is not an invalid state.

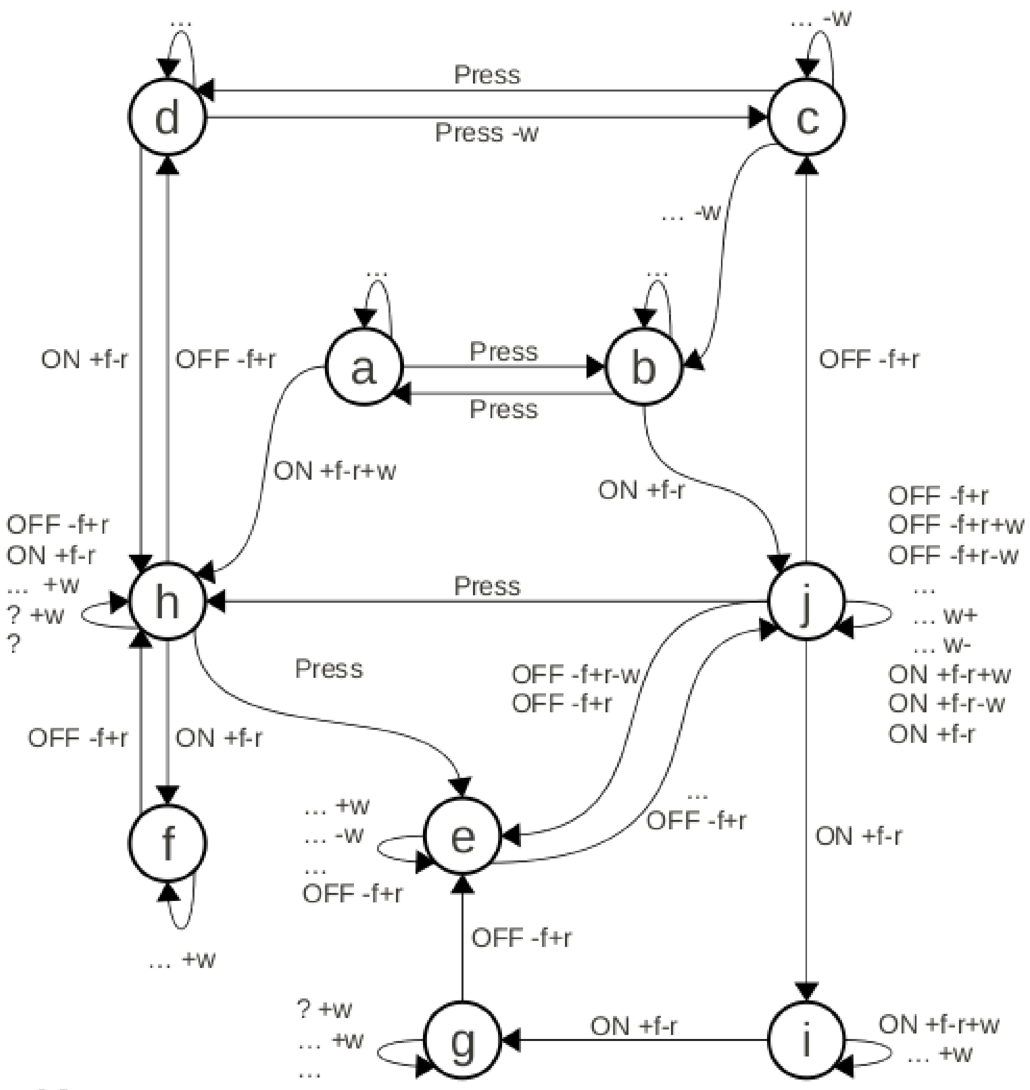
noticing its position relative to other objects for example). The use of a simulated vision system is given extra justification because it makes subtle perceptual mistakes that real systems also make.

The system in figure 5.1 is just one instance of a variety of different behaviours that the agent may observe when it pushes the ball. At times the agent will observe the 'clean' version and at others it will observe some other subtly different behaviour. However, the observed behaviours will all be consistent in the sense that the ball will always transition through the moving state and eventually return to the stationary state. An automated procedure could be developed to remove predictable unusual states (such as state **b**) and this may simplify the learning task.

Tap and Sink Example

The tap and sink system is an example of a more complex history generated from the tap and sink simulation (the tap and sink system is described in section 4.5.1). The system behaviour is shown in figure 5.2; it has a total of 10 states and 49 transitions. The states of the system are described in table 5.2. The behaviour graph was generated by turning the tap lever anti-clockwise and clockwise ('ON' and 'OFF'), and by pressing the button that controls the plug. Note that the state of the plug can be configured to be observable or hidden. In this instance the state of the plug *is* observable which leads to fewer ambiguous transitions, however there are still a significant number of these present.

State **j** is the state in which the tap is on, the plug is open, and the water level is relatively steady (the outflow from the plug is approximately equal to the inflow from the tap). A large number of transitions emanate from this state, 14 in total. The only unambiguous action is to insert the plug



Key

ON	Turn tap lever anti-clockwise to increase flow.
OFF	Turn tap lever clockwise to decrease flow.
Press	Press the plug button to insert or remove plug.
…	No action (time passes).
?	Multiple actions observed.
f	Tap flow rate qualitative delta.
r	Tap lever rotation qualitative delta.
w	Water level in sink qualitative delta.

Figure 5.2: Observed Tap and Sink Behaviour

State	Description
a	Sink is empty, plug is in.
b	Sink is empty, plug is out.
c	Sink is emptying.
d	Sink has water, no inflow or outflow.
e	Sink is emptying, tap is on.
f	Sink is filling, tap is *fully* on.
g	Sink is filling, tap is *fully* on, plug is out.
h	Sink is filling.
i	Sink is filling, plug is out.
j	Water level is steady, tap is on, plug is out.

Table 5.2: Observed Tap and Sink States

which moves the system into a state in which the water level is increasing. The remaining actions, increasing or decreasing flow and 'time-passes', are all ambiguous. The time-passes transitions are all self-loops which indicates that state **j** is stable (i.e. it will not spontaneously transition to a new qualitative state). However three different sets of deltas are observed coinciding with the time-passes transition. The water level is observed as rising or decreasing by some amount or remaining level. This is expected behaviour because the inflow and outflow vary slightly over time and are rarely exactly equal. At any given time in this state the water level is normally slightly rising or falling.

The decrease flow action ('OFF') from state **j** has various different observed outcomes. The transitions which return to state **j** are similar to those of the time-passes transitions: the flow rate always decreases and the rotation of the tap lever always increases, but the change in water level has various outcomes (increase, decrease or stay the same). In this case however the water level is far more likely to decrease (because the inflow has decreased) and therefore this will become the expected transition once the agent has observed enough instances to identify the trend. The other transitions caused by the OFF action are as might be expected: a transition to state **c** (where the level is falling) occurs when the inflow has been cut off completely, a transition to state **e** occurs if the inflow is reduced (but not cut off) such that the water level is now falling rather than steady (i.e. the qualitative derivative of the water level is less than 0).

As can be seen from the two examples presented in this section observed histories can uncover some unexpected states and transitions in the behaviour of relatively simple systems. These are born of a combination of imprecision in the vision system and the ambiguity inherent in qualitatively abstracted action outcomes. Many of the more unusual transitions occur rarely and as such are candidates for removal from generalised models, assuming the agent has made a sufficient number of observations to do so.

The examples also show the necessity of including qualitative deltas in action descriptions. Without the deltas a number of the actions would have no observable effect. Furthermore, a representation in which actions do not have qualitative deltas would have to model actions at a higher level of abstraction (e.g. 'turn the tap clockwise' would have to become something similar to 'turn to position X' where X is some pre-determined interesting concrete value). The deltas allow otherwise identical actions to be distinguished without resorting to quantitative techniques.

5.3 History Generating Algorithm

The purpose of the history generating algorithm is to create a sequence of histories from a sequence of snapshot observations in real-time. The snapshot observations are produced by the agent's vision system. The generated histories will be used to create generalised models of observed behaviour. The algorithm is used in a pipeline in which the snapshots arrive asynchronously (i.e. at any time) and completed histories are passed on to one or more listeners, also asynchronously. An outline of the history generator is specified in Algorithm 1.

Algorithm 1 GenerateHistories

```
$histories \leftarrow \emptyset$
loop
  $observations \leftarrow$ get_snapshot()
  if start_new_history( $observations$ ) then
    add new_history() to $histories$
  for $h$ in $histories$ do
    if stop_history( $h$, $observations$ ) then
      alert_listeners( $h$ )
      remove $h$ from $histories$
    else
      $newObjects \leftarrow$ find_new_objects( $observations$ )
      update_with_new_objects( $h$, $newObjects$ )
      $newTransition \leftarrow$ filter( $observations$, $h.objects$ )
      add $newTransition$ to $h$
```

GenerateHistories continuously processes observation snapshots while maintaining and updating a set of active histories. For each snapshot a check is made to see if a new history should be started. A history can be started for a variety of reasons; this might include observing a significant change in the environment or an instructor explicitly signalling.[2] If a new history is required it is added to the set of active histories. The algorithm then proceeds by iterating through each active history. A check is made to see if the history should be stopped at this point. Again there are a variety of reasons why this might be the case, for example, no new behaviour has been observed for a significant amount of time. If the history is stopped

[2]*GenerateHistories* uses only the current set of observations to make a decision on starting a new history. It could be adapted to use other information, such as the current set of active histories, if a particular heuristic required it.

then it is removed from the set of active histories and sent to any listeners (typically the system generalizer). The start and stop functions control the temporal context of the histories produced by the generator.

If a history is not stopped then it gets updated. This involves expanding the context with any new 'relevant' objects and adding a new transition created from the latest snapshot. There are different options for determining which objects are added to the history's context. For example, one way is to add any objects which have recently changed to the context.

Adding new objects to an active history's context requires the history's *existing* transitions to be updated with information about the new objects. This is trivial if it is possible to go back in time and extract the object information from the relevant observation snapshots, however this is not necessarily possible so some assumptions may be required to fill-in the new objects' prior state.

Finally a new transition is added to the history. It is created by filtering the latest observation snapshot so that it only includes assertions about objects in the history's context.

5.3.1 History Generation Heuristics

The algorithm described in the previous section provides an outline of the history generator. As has been noted there are a variety of options for implementing key steps in the process and these have not yet been described in detail. For example, the test for when an active history should be stopped and removed from the set of active histories. Each of these functions plays an important role in determining the types of histories that are generated. The ability of the generalizer to find good models will depend on how these heuristic functions are implemented. This section describes the heuristics of the generator and the different options for implementing them.

Temporal Restriction: Starting New Histories

This function is called for every snapshot and returns true if a new history should be started. It implements part of a history's temporal restriction. The following methods are used to determine if a new history should be started:

- Every Snapshot: a simple implementation is to start a new history on every new snapshot. This has the advantage of avoiding the problem of missing useful transitions. However it can be computationally intensive, especially if histories are quite long (in which case there will be many histories active simultaneously).

- Always Exactly One: an alternative is to ensure that there is always a single active history by only starting a new one when the previous one finishes. Like the 'Every Snapshot' approach this approach ensures that no potentially useful transitions are missed, however it does preclude simultaneous active histories (which may have differing contextual restrictions, i.e. focused on different sets of objects).

- Wait for Event: this method starts a new history whenever something 'interesting' is observed. An interesting event may be the agent performing an action or it might be a change in the environment involving multiple objects. This method has the advantage of only starting histories when there is something worthwhile to learn. Potentially this makes the job of the generalizer easier since it has fewer histories to learn from. However, it has the disadvantage that useful transitions can be missed. This trade-off is determined by the precise definition of an interesting event.

- Teacher Initiated: a signal from the teacher can be used to start a new history. In this case the agent is relying on the teacher to ensure that no interesting behaviour is missed. The advantage to this method is that histories should be more useful training examples. The disadvantage of this approach is that the agent loses some autonomy.

Temporal Restriction: Stopping Histories

This function is called for every active history at every snapshot to determine if the history should be terminated. It implements part of the history's temporal restriction. The following methods can be used to determine if an active history should be terminated:

- Fixed Time Limit: this method stops the history after a fixed duration. It is a trivial method of preventing histories from becoming too long. The length of time for which the history is active can be measured in either snapshots or action executions. For example, terminate the history if the agent has executed 10 actions since the history was initiated.
- Limit Size of History: stop the history once it has reached a certain size. The 'size' of a history can be measured in the following ways: the number of transitions, the number of states, the number of cycles, or the number of objects. This method has the advantage that the structure of histories can be made to be similar to the desired structure of generalised models (assuming there is a desirable structure for generalised models). Note that restricting the history size to a *single* transition results in a dataset that is similar to those used in many existing action learning systems (see section 2.2).
- No New Interesting Observations: stop the history if no new 'interesting' observations have occurred for a fixed duration. Interesting observations are those that add something new to the history. These include: transitions to new states, new transitions between previously observed states, and the addition of new objects to the context of the history. Potentially this method can make learning more efficient by automatically stopping histories when an observed system

has been 'explored' to the extent that observing new behaviour is unlikely. This is dependent on the particular system being observed and the precise definition of what is an interesting observation. The duration for which no new interesting observations are observed can be measured in elapsed actions or snapshots.

- Teacher Terminated: a signal from the teacher can be used to terminate a history. This has the advantage that the teacher can steer learning towards desired goal systems by limiting histories to relevant episodes of agent interactions. The disadvantage of this approach is that the agent loses some autonomy.

Another possible heuristic 'context failure' uses a partially learned model when deciding to stop a history. In this case the history is stopped when the context of the learned model is no longer applicable to (satisfied by) the current world state. This would enable histories to discover new states not included in the model and automatically stop when this is no longer possible. The context failure heuristic is not implemented in this experiment because the focus is on stand alone history generation, however it would make a good candidate for an integrated learner and history generator.

Contextual Restriction: Adding New Objects

The contextual restriction of a history determines which objects are considered a part of the system under observation. A history only contains information about objects in its context and disregards the rest. The context is constructed by checking each snapshot to see if any new objects should be added. The following methods can be used to determine the result of the 'find_new_objects' function and hence which objects should be added to the context for a given history:

- All Objects: assume all observable objects are relevant to the system being observed. This approach is only feasible in trivial worlds with only a few observable objects. It is useful for generating complete histories from a micro-world which can be used for comparison with the contextually restricted histories generated from complex worlds.

- Any Objects That Change: if an object changes then it gets added to the current context. This approach ensures that anything directly affected by the observed system will definitely be included in the history. It has the disadvantage that objects affected by events exogenous to the system will also be included. Objects are considered to have changed if a property changes or a relationship with another object changes.

- Any Relevant Objects That Change: if an object changes *and* it is in some way relevant to the current context then it gets added to the current context. This approach addresses the issue of avoiding adding objects to the context which have been affected by exogenous events. It does so by using a relevancy test to determine if a changing object is likely to be part of the system under observation. The relevancy test is implemented in various ways, for example, objects are considered relevant if they are in close proximity to an object already in the system context. Stricter tests include requiring that the object is touching or connected to a context object. A disadvantage of this approach is that a poorly chosen relevancy test will exclude important objects from a system history.

Updating Existing Transitions

The history generator constructs histories incrementally by adding new transitions as they are observed. When a new transition contains new objects to be added to the history's context then the history's existing transitions must be updated with the state of the new objects. This ensures that histories are complete descriptions of observed systems over some time period. The following methods can be used to update existing transitions with new context objects:

- Use Buffer: buffer sufficient snapshots such that the state of objects in previous transitions can be found by inspecting the buffer. This method ensures that the previous state of objects is correctly recorded. The disadvantage of this approach is that it is only feasible if certain unrealistic restrictions are made on the vision system and simulation.
- Assume Not Changed: assume that objects not previously included in the context have not changed since the history began. This approach is implemented by copying the object state from the most recent snapshot to the existing transitions. A problem with this approach is that the assumption that new objects have not changed may not be true. This depends on the rules used to add new objects to the history. If they have indeed changed then the history will contain false information on the state of some objects. This problem is avoided when using the 'Any Objects That Change' method for finding new context objects.
- Leave Unknown: leave values for objects in previous transitions as unknown. This approach has the advantage that it avoids making any false assumptions about object states. It has the disadvantage that the history will contain transitions with ambiguous states.

5.4 Effect of Heuristics on History Construction

This section describes the results of experimenting with the history generator's heuristics. The purpose of the experimentation was to discover the general properties of the datasets generated by the GenerateHistories algorithm and use this information to assess their suitability as input to a generalising algorithm. The algorithm's heuristics described in the previous section are varied and their effects observed.

5.4.1 Generating Datasets

Datasets are sets of histories output by the GenerateHistories algorithm. They are created by starting the simulation, executing actions, and then observing the objects in the environment. The resulting observations are then used as input to the GenerateHistories algorithm. Each run is specified by the duration the simulation is run for, a strategy for choosing actions, and finally the heuristics used.

For each dataset the simulation is run for a fixed duration measured in terms of observation snapshots. The longest runs are 1000 snapshots which is sufficient to execute over 100 actions and for many different object interactions to be observed. These runs take approximately 90 minutes to execute; the majority of this time is taken up executing actions which typically take 1 to 10 seconds to complete (note that the vision system is paused while actions are executed to preserve their atomicity property, see section 4.3).

The precise actions chosen by the agent will affect the output of the HistoryGenerator. For example, if the agent executes the same action on the same object over and over, then the resulting histories will be very similar. In this experiment, two action strategies are used in generating the history datasets. The first is to choose actions randomly from a restricted subset of action types and target objects, thus allowing the agent to explore a particular domain. The second strategy is to allow an instructor to execute all actions. The instructor chooses actions which they feel demonstrate a particular mechanism or mechanisms.

The heuristics are varied for each run to observe their effect. The four heuristics are: the history starter, history stopper, the context updater, and the existing transitions updater.

5.4.2 Results

This section presents the results of running the simulation with various combinations of heuristics. The purpose of the experiment to establish which combinations of heuristics are most likely to generate examples of the types systems that the agent will find useful. The toy world environment was used for this experiment (see section 4.5.2). In this particular environment the agent should be able to learn simple systems that describe basic physical interactions (such as pushing a ball) and also more complex systems involving multiple objects (such as building a tower of blocks). To learn these types of models the generated examples must have a structure that is similiar to the these models.

Results were assessed by calculating various properties of the generated histories and also by manually inspecting the resulting histories. The following properties were calculated for each run:

- Total number of histories generated. A sufficient throughput of histories is required for the agent to learn effectively. A low number indicates that either the histories are too large or that useful obsevations are being discarded.

- Minimum, median and maximum size of histories (measured in number of states). Models generated from the toy world should not be greater than about 10 states. More complex systems will be difficult to apply to new situtations.

- Average transition density (transitions per state). A higher transition density indicates that the behaviour of systems is being more thoroughly explored. This is good for learning as fewer histories will be required to learn a complete model of behaviour.

- Average number of objects in the context. If the average number of objects is very high then it indicates the history generator is exploring systems that are too large. Histories with more objects are less desirable because they are less likely to be applicable to other situations.

- Average number of assertions in the context. The context assertions are used to create the context of the generalized models. A small value may indicate that important constraints are not being recorded in the histories.

- Total number of histories containing at least 1 cycle. Many useful systems are cyclic, therefore a higher number of histories with cycles is desirable.

Table 5.3 shows the results of varying the heuristic used to stop active histories. The simulation was run for a duration of 1000 observations and the agent executed random actions from a limited set. The agent executed around 260 actions during each run. The other heuristics were fixed at the following values: *starter* - maintain at least 1 active history unless nothing is happening; *context updater* - add any objects that change to the context; *states updater* - assume new objects have not changed when updating existing states. The results were averaged over 3 independent runs.

Four different mechanisms for stopping histories were tested. These were: stopping after a fixed number of snapshots; stopping after a fixed number of actions; stopping the history if it contained a cycle (i.e. returned to a previous recorded state); and finally, stopping if the history's context reached a certain number of objects.

Table 5.4 shows the results of varying the mechanism used to start new histories. Two mechanisms were tested: ensuring that a single history was always active (by starting a new one as soon as the active history finishes); and ensuring that a single history was always active *unless* nothing is happening (i.e. don't start a new history if there are no changes or actions currently occurring).

Table 5.5 shows the results of varying the mechanism used to add new objects to the history's context. Two mechanisms were tested: add any ob-

		States						
History Stopper	**Histories**	**Min**	**Med**	**Max**	**Density**	**Objects**	**Context**	**Cycles**
after 10 snapshots	82	1	5	11	1.37	8.8	110	32
after 20 snapshots	45	4	9	18	1.51	13.1	191	29
after 5 actions	52	3	8	19	1.43	12.1	170	30
after 10 actions	26	9	15	33	1.55	17.5	281	22
limit 1 cycle	53	2	6	49	1.41	9.5	131	53
limit 3 objects	215	2	2	5	0.75	4.9	48	25
limit 5 objects	147	2	3	7	0.95	6.6	72	28

Table 5.3: Results of Changing the History Stopper

		States						
History Starter	**Histories**	**Min**	**Med**	**Max**	**Density**	**Objects**	**Context**	**Cycles**
always 1 (if event)	82	1	5	11	1.37	8.8	110	32
always 1	90	2	5	10	1.35	8.2	101	33

Table 5.4: Results of Changing the History Starter

jects which change to the context, and add any objects which *qualitatively* change to the context (i.e. ignore non-qualitative changes).

		States						
Context Updater	**Histories**	**Min**	**Med**	**Max**	**Density**	**Objects**	**Context**	**Cycles**
any changed objs.	82	1	5	11	1.37	8.8	110	32
ignore non-qual.	83	1	5	11	1.39	8.5	109	33

Table 5.5: Results of Changing the Context Updater

Table 5.6 shows the results of varying the mechanism used to update existing states when the context is expanded. Three mechanisms were tested: assume that the new objects have not changed until they were added to the context; use a vision system buffer to fill-in the correct state of the objects in prior states; and finally, leave the state of the new objects unknown in previous states.

		States						
State Updater	**Histories**	**Min**	**Med**	**Max**	**Density**	**Objects**	**Context**	**Cycles**
assume no change	82	1	5	11	1.37	8.8	110	32
use buffer	83	1	5	11	1.38	8.7	110	33
leave unknown	82	2	8	15	1.02	8.8	27	30

Table 5.6: Results of Changing the Existing States Updater

5.4.3 Analysis

This section describes some of the interesting results that can be observed from experiments with the history generator. Overall the results give a good indication of the kinds of histories that can be generated from an autonomous agent making symbolic observations of a 3D environment. This information can be used to guide the choosing of a history generator's heuristics when it is being used in conjunction with a system generalizer.

The results from the experiments were very consistent. The data tables show the average values for each test over 3 independent runs. Each of the 3 runs uses identical heuristics with the only change being the actions which were chosen by the agent. Despite each run having different actions (and therefore very different sequences of events), there was very little variation in the measured values between runs. This can be attributed to the tests being executed in a single domain (the toy world) with similar objects and hence the same types of sequences of object behaviour were observed. However there was significant variation when the generator's heuristics were varied.

The 'history stopper' heuristic was varied the most with 7 different values tested (table 5.3). As expected, the number of histories generated was inversely proportional to the size of the histories themselves. Results ranged from 26 large histories to 215 small histories. This ratio can be directly controlled using the 'after X snapshots/actions' history stopper. The larger histories not only had more states but also more objects. Ideally histories will be an almost complete set of transitions describing the behaviour of a small number of objects. This will aid the generalizer because it will require fewer histories to build a complete picture. Unfortunately the results show that larger histories are also correlated with more objects which indicates a better strategy for choosing actions is required.

The median number of states varied from 2 to 15 per history. An 'ideal' number of states will vary depending on the target models and on the richness of the observations. The 'push ball' system (see section 5.2) requires only 3 states compared with 10 or more required by the 'tap and sink' system. Also, a richer set of qualitative observations will lead to more detailed sequences of behaviour with more states. Given this variation it seems that the *maximum* history size is more important than the typical size. Ensuring an appropriately high maximum will prevent histories from being cut short before a system has been fully explored.

The completeness of a history's description of a system behaviour is given not only by the number of states but also by the number of transitions. This is because there can be multiple transitions between the same two states. The completeness of history is roughly measured using transition density (the number of transitions per state). Varying the history stopper heuristic did not have much effect on transition density unless the 'limit to X objects' strategy was used. This dropped the density from around 1.5 down to less than 1. Visual inspection showed that the combination of a few objects and random actions resulted in histories where states were not revisited and hence each new transition also led to a new state.

Cycles is histories indicate that a previously visited state has been returned to. Cycles are desirable in histories because most useful systems, especially more complex ones, will contain at least 1 cycle. Setting the stopper heuristic to halt after a cycle is discovered resulted in the most histories with cycles. Otherwise, the percentage of cycles was roughly correlated with the size of the histories produced. Larger histories were more likely to find a cycle. Many of the cycles involved just 2 states and some of these were the result of the vision system noticing a 'wobble', e.g. a non-moving object may be momentarily mistaken as moving. A better cycle metric for histories would eliminate these anomalies as they are not indicative of a system-like cycle of behaviour.

The largest history generated by the cycle limited stopper had 49 states, which is far larger than those generated by any of the other stopping mechanisms. This indicates that it might be useful to include an abort mechanism with the cycle limiter to prevent histories from becoming too large. A 49 state system is unlikely to be a useful system because it will be more difficult to apply it to new situations.

Two history starters were compared. One ensures a single history is always active, the other only starts a new history if there is no active history and an 'interesting' event occurs (i.e. something qualitatively changes in the environment). The histories generated are very similar (see table 5.4), the only difference is that the interesting event starter generated approximately 10% fewer histories over the same period. This is because a fixed length history stopper was used and the generator was active for a fewer overall number of snapshots. This reduction could be used to improve the computational performance of the learning system by reducing the number of histories that must be processed.

The context updater is used to determine which objects get added to a history's context. Two context updaters were compared, one adds objects which change in any way, the other adds objects which change qualitatively only (i.e. it ignores qualitative deltas, see section 3.2.3). Very little difference can be observed in the types of histories generated using these two updaters. This is because objects which change non-qualitatively will soon change, or have recently changed, qualitative state.

The existing state updater is used to fill in the previous state of objects newly added to a history. The 'assume no change' and 'use buffer' techniques resulted in the output of similar histories. The 'leave unknown' technique resulted in larger histories with smaller contexts. The increase in states occurred because a variable with an unknown value is treated as a distinct state from a variable with a fixed value. Therefore returning to a previously visited state *with new objects in the context* is not recognised as the same state since the history's recorded state is ambiguous with regard to the new context objects and therefore cannot be matched to the currently observed state.

Ideally the generalizer would learn from unambiguous and correct histories. The buffer approach to updating existing states is the only technique to guarantee this but it makes unrealistic assumptions about the vision system. These assumptions can be side-stepped in the simulation but not in a real embodied agent.[3] Therefore the realistic options for generat-

[3]An embodied agent can record large amounts of raw visual input but it cannot go back in time and move its head or change its focus to an object on which it wasn't focussing at the time.

ing histories are to either assume the object has not changed, or to leave the objects previous state as ambiguous. The former introduces errors (which could be considered a type of noise) into the histories; the latter requires the generalizer to directly resolve the ambiguities.

Overall the results show that suitable histories can be generated from a complex simulated environment. A number of effects of changing the history generator's heuristics have been observed and their potential impact on a system generalizer have been assessed. It is not clear what precise combination of heuristics will lead to successful learning. This can only be assessed in conjunction with the actual system generalizer and by directly observing its performance. However the results obtained so far have provided a better understanding of the process of generating histories and therefore of how to configure the generator to produce the particular types of histories which may be desired. This in turn will aid the process of discovering a generalizer that produces good system models.

The results have also shown that a random action strategy produces histories that can be used for learning small systems. However, it is clear that autonomous learning of *larger* systems will be almost impossible unless a heuristic strategy that ensures actions are relevant to the system at hand is used (or alternatively a teacher is used to guide the agent's actions).

Chapter 6

Using Q-Systems in Learning & Planning

This chapter describes an implementation of the Q-System representation in an autonomous agent system that performs basic learning and planning. The primary purpose of the system is to demonstrate the use of Q-Systems as a suitable representation for learning complex world dynamics and action outcomes. A second purpose is to discover any limitations or issues that would need to be addressed by a more sophisticated agent. The algorithms described in this chapter are not considered robust with respect to their application in a complex environment, however they do enable the agent to learn simple generalised models and also to achieve basic goals.

The algorithms include a matcher which can map the objects and states in one system to those of another, a learner which generalises example systems, and a planner which chooses actions that lead the agent towards a specified goal. Section 4 of the chapter describes an integrated implementation of the algorithms followed by a demonstration of its application in the simulated environment. The last section discusses the major limitations of the algorithms.

6.1 Matching

Matching is used in planning to map abstract models to concrete world states, thus enabling an agent to predict the behaviour of matched objects on the basis of knowledge in the model. Matching is also used in model refinement to map abstract models to other abstract models. If the models are a good fit they may be considered different instances of the same system and subsequently merged and generalised.

Matching involves finding a good mapping between the objects in one description and the objects in another. Each individual pairing of object to object specifies that the two objects play the same role in the two descriptions.

Finding a good match can be considered a type of search problem in which the search space is all possible mappings of objects. Naive search is not practical because the search space grows exponentially with the number of objects. Therefore, the algorithm presented here uses a score based heuristic to find good *individual* object pairings (i.e. disregarding the constraints that the pair may place on other possible object pairings). The set of individual scores can be calculated in polynomial time which is practical for the numbers of objects typically used in Q-Systems.

Scores are based on a similarity metric which considers the properties, relationships and behaviour of the two objects being compared. Objects with more in common will score higher than those with less. Once all object pairs have been scored, constructing a complete match requires finding the highest scoring set of pairs.

An important consideration in matching is whether an object can be mapped to exactly one object or may be mapped to more than one object or mapped to none at all. An 'unconstrained' matching allows objects on either side of the match to be paired with any number of objects, i.e. a 'many-to-many' relationship. Conversely, a fully constrained matching ensures each object is only paired with one other, 'one-to-one'. A semi-constrained matching allows objects on only one side to match multiple objects on the other, 'one-to-many'. When an object is mapped to multiple objects the match represents the object playing multiple roles. In a good matching the object should play the role of *all* the objects it is mapped to simultaneously.

Matching can be performed on both individual states and entire systems. Individual state matching is used when matching a system context to a concrete world state. Whole system matching is used when matching models to histories. Matching of systems requires finding not only a match between objects but also a match between two systems' states. This results in a search space that is exponential in both the system objects and the system states. The approach used in the following algorithm simplifies the problem by recognising that an object matching heavily constrains the number of possible state matchings.

6.1.1 System Matching Algorithm

The section describes the algorithm used to match systems to other systems for use in the system generalizer. State to state matching uses a simplified version of the same algorithm. Pseudo-code for the system matcher is shown in Algorithm 2. The goal of the algorithm is produce a 'good' (rather than best possible) matching of objects and states. The matcher has two main stages: firstly, finding a good object match between the two systems' contexts; and secondly, finding a consistent state match based on the object mapping.

The matcher begins by scoring all possible pairs of individual object matches. Pairs of objects are scored by their similarity in the system context and by the similarity of the transitions in which they are involved. The two parts of score are normalised (in the range 0 to 1), then weighted (context can be given more importance than behaviour or vice versa), and finally added.

Algorithm 2 MatchSystems(src, tgt)

```
// Find a good object matching between system contexts...
$objMatch \leftarrow$ empty mapping of $src$ to $tgt$ objects
$candidatePairs \leftarrow$ apply score function to elements of $src.objects \times tgt.objects$
sort $candidatePairs$ by score
for ($srcObj$, $tgtObj$, $score$) in $candidatePairs$ do
   if $srcObj$ not matched then
      if all $tgt.objs$ are matched then
         add ($srcObj$, $tgtObj$) to $objMatch$
      else if $tgt.obj$ is not matched then
         add ($srcObj$, $tgtObj$) to $objMatch$
until all $src.objects$ are matched

// Match identical states...
$stateMatch \leftarrow$ empty mapping of $src$ to $tgt$ states
for ($srcState$, $tgtState$) in $src.states \times tgt.states$ do
   if substitue($srcState$, $objMatch$) = $tgtState$ then
      add ($srcState$, $tgtState$) to $stateMatch$

return ($objMatch$, $stateMatch$)
```

The next step is to create a mapping by selecting the highest scoring pair for each object in the source system and adding the pair to the match. The algorithm will create a one-to-one mapping if possible (when the target has the same number or more objects than source), otherwise it will create a many-to-one mapping by pairing the additional source objects to already matched target objects.

After finding an object mapping the matcher creates a state mapping. This is achieved by finding states that are identical when the object mapping is used to substitute variables in the source system. Because the matcher only creates either fully or semi-constrained mappings the substitution for the source objects is unambiguous. If the mapping were allowed to be unconstrained (many-to-many) then the substitution would be much more problematic.[1]

Finally, the result is returned as a tuple containing the object and state match.

6.2 Generalising

This section describes a basic algorithm for learning generalised Q-Systems. The purpose of the learning system is to transform a series of observed example concrete systems into a set of generalised systems that are applicable to analogous situations. This enables the agent to use its experience of the world to predict and plan for the behaviour of objects in new and unfamiliar situations.

A good learning algorithm will produce models that are not over generalised. Over generalised models will apply to many situations but will not correctly predict behaviour. Similarly a good learner will not under generalise by missing opportunities to merge examples describing similar systems.

[1]For example, how should a substitution be resolved if a source object is paired with two target objects that have conflicting assertions?

The core of the learning algorithm works by matching systems that have similar contexts and replacing them with a new system. The new system *context* is the intersection of the two contexts - the assertions that are common to both systems, however, the *behaviour* of the new system is the union of the behaviour of both systems (the learner assumes that behaviour observed in one example and not the other was missed and could have been present in both examples given enough observations). The learner applies a combination of operations to convert a system including: replacing concrete objects with abstract variables, dropping assertions from the context, adding states to the behaviour, and, adding transitions to the behaviour. This is described in section 6.2.1.

Before the learner can generalize the systems it must select which systems are to be generalised. The algorithm 'LearnSystems' determines the systems that should be generalised and constructs a match between them for use in model refinement. It is described in section 6.2.2.

6.2.1 Model Refinement Algorithm

The purpose of the model refinement algorithm is to create a generalised system from an existing model system and a newly observed example system (a history). The algorithm takes three inputs: the model, the history, and a matching from one to the other. The matching is assumed to be the best available matching between the two systems. Pseudo-code for the model refiner is shown in Algorithm 3.

The algorithm starts by using the object match to find a substitution for each input system that maps objects to the new general system. This is a form of 'anti-unification' in which paired objects are replaced by new variables (unless both objects are the same constant in which case they are not replaced by variables). The resulting 'unifier' contains two mappings of objects, one for each input system. Next, the new system's context is calculated by finding assertions that are common to both the example and

Algorithm 3 RefineModel($model, history, match$)

```
$unifier \leftarrow$ antiunify( $match.objMatch$ )
$context \leftarrow$ union( $model.context/unifier, history.context/unifier$ )

$behaviour \leftarrow \emptyset$
**for** ($s_{model}, s_{history}$) **in** $match.stateMatch$ **do**
    add union( $s_{model}/unifier, s_{history}/unifier$ ) to $behaviour$
**for each** unmatched state $s$ **do**
    add $s/unifier$ to $behaviour$

**return** new_system( $context, behaviour$ )
```

model contexts, given the substitution.

The behaviour of the new system is built by iterating through the transitions in both the model and example systems. If a state in one system is matched to a state in the other, then the two states are generalised in a similar way to the context generalisation. If a state is not matched, then it is simply added to the new system. All transitions in both systems are added to the new system.

Finally, the new system with the generated context and behaviour is returned.

6.2.2 Model Selection Algorithm

The purpose of the model selection algorithm is to process an endless series of example systems ('histories') and maintain a knowledge-base in the form of a set of generalised systems. The algorithm uses the RefineModel program described in the previous section. Pseudo-code for the learner is shown in Algorithm 4.

The learner has two parameters: a set of weights for use in system similarity scores (these can be used to change the relative importance of transition, state and context similarities), and a threshold for use in deciding whether or not two systems are similar enough to be generalised (the threshold can be used to make the learner a more or less 'eager' generalizer).

Algorithm 4 LearnSystems($weights$, $threshold$)

```
$systems \leftarrow$ null
**loop**
    $history \leftarrow$ get_next_history()

    // Match and score existing systems to new system...
    $matchings \leftarrow$ apply MatchSystems to each system in $systems$
    $scores \leftarrow \emptyset$
    **for** $s$, $m$ **in** $systems$, $matchings$ **do**
        $score \leftarrow$ (transition_score( $s$, $history$, $m$ ) $\times$ $weights.transition$) +
            (state_score( $s$, $history$, $m$ ) $\times$ $weights.states$) +
            (context_score( $s$, $history$, $m$ ) $\times$ $weights.context$)
        add ($score$, $s$) to $scores$

    // Refine best match and add to knowledge base...
    **if** $scores.max > threshold$ **then**
        $newSystem \leftarrow$ RefineModel( $scores.best$, $history$ )
        remove $scores.best$ from $systems$
        add $newSystem$ to $systems$
    **else**
        add $history$ to $systems$
```

The learner begins with an empty set of systems. It then enters an infinite loop and waits for new histories. For each new history the learner finds a matching and a similarity score (based on the matching) for each of the systems in its knowledge base.

If the highest scoring match is above the generalising threshold, then the best matched system and the history are generalised to create a new system. The new system replaces the existing system. If on the other hand, the highest score is not above the threshold, then the example itself is added to the knowledge base. This is done in the expectation that it will at some point be generalised.

Similarity scores are calculated by combining scores for the systems' transitions, states and the contexts. Transitions count towards the score if both ends (i.e. the pre and post states) are matched *and* they are the same action type. Matched states are scored for each assertion common to both systems. System contexts are scored according to the *fraction* of assertions that are common to both systems.

6.3 A Planner

This section describes a simple planner that uses Q-Systems to achieve goals. The purpose of the planner is demonstrate how Q-Systems can be used to achieve goals. The planner attempts to find and execute a series of actions that will change the current state of the world to one that satisfies a set of goal assertions. The planner is given a goal which is typically a small number of assertions (and therefore a partial description of a world state). Pseudo-code for the planning algorithm is shown in Algorithm 5.

The planner works by searching its knowledge base for a Q-System that matches the current world state *and* contains a reachable state which satisfies the goal. It then prunes the system of all states from which the goal state cannot be reached. The resulting pruned system is then used to choose which actions to execute. The planner only executes plans based on a single system; it cannot chain multiple systems together to form more elaborate plans.

Once a pruned model has been found the planner executes actions that lead towards the goal state (i.e. actions that reduce the distance between

Algorithm 5 AchieveGoal($goal$, $knowledgeBase$)

```
$model \leftarrow$ null
loop
  $observation \leftarrow$ get_snapshot()
  if goal not achieved then
    if $model = null$ then
      // Try to find a suitable model...
      for $system$ in $knowledgeBase$ do
        $match \leftarrow$ greedy_search_matching( $observation$, $goal$, $system$ )
        if $match \neq$ null then
          $pruned \leftarrow system.removeStatesNotOnPathToGoal()$
          $model \leftarrow (pruned, match)$
    else
      // Execute an action to get closer to the goal...
      $model.updateMatching(observation.newObjects)$
      $currentState \leftarrow model.findCompatibleState(observation)$
      if $currentState =$ null then
        // The world is in an unexpected state so find a new model...
        $model \leftarrow$ null
      else
        $goalState \leftarrow model.findCompatibleState(goal)$
        $action \leftarrow model.shortestPath(currentState, goalState).head$
        execute_action( $action$ )
```

the current and goal state, where 'distance' is the minimum number of actions to get from one to the other). The planner continuously executes actions until the goal is reached, at which point it stops and waits for a new goal. If at any stage the agent observes a world state which is not compatible with the state(s) predicted by the pruned model, then the planner discards the current model and 're-plans' by searching the knowledge base once more.

One property of Q-Systems is that the outcomes of actions are often ambiguous. This may be because the action itself is inherently non-deterministic (e.g. throwing a ball into a bin will sometimes land in the bin and sometimes not), or the ambiguity may be the result of qualitative abstraction[2]. The planner assumes that the outcome with the highest probability will occur when calculating shortest paths from the current state to the goal. Then, while executing, if by chance the actual outcome is not the most likely outcome then the planner simply continues from the state it is now in.

Another property of Q-Systems is that some system objects only exist in *some* of a system's states, since objects are sometimes created or destroyed by actions within the system. The presence of these objects forces the planner to recompute its model-to-world object matching whenever new objects appear in the world, but does not affect the planning algorithm.

6.4 Learning & Planning in the Simulated World

This section describes an integrated implementation of the algorithms described in the preceding sections. The algorithms are used to create a learning and planning agent that exists in the simulated toy world; the

[2]This is because a single qualitative value represents a whole range of real values, but the outcome of an action may depend on the precise value of the variable. Exact (quantitative) values cannot be observed directly and therefore the outcome is ambiguous.

agent interacts with the world using the skill and vision systems described in previous chapters. The system learns simple generalised models from examples and uses them to achieve instructor specified goals. The question of how the quality of the models output by the learner is measured is open-ended. In this project a combination of task performance and manual assessment is used.

6.4.1 Agent Implementation & Performance

The agent has two modes: a learning mode and a planning mode. The modes are mutually exclusive so at any one time the agent is either learning or planning but not both (note, a more sophisticated agent might learn from experience while simultaneously executing a plan). While in the learning mode the agent builds its knowledge base of Q-System models using the learning algorithm described in section 6.2. During the planning mode the agent accepts goals from the instructor and attempts to achieve them using the planning algorithm, described in section 6.3.

The learning system is implemented using a pipeline architecture in which each component receives input (asynchronously) from its predecessor. The input is then processed and the output is sent to the next component in the pipeline. The learning system has 4 components in its pipeline: the simulation, the vision system, the history generator, and finally the generalizer. The simulation generates raw simulation data based on physical interactions of objects. The vision system takes the raw simulation data and extracts qualitative observation snapshots. The history generator creates histories from sequences of observation snapshots. Finally, the generalizer uses histories to update its knowledge-base of model systems.

The agent was run in the simulated toy world and was set the task of building a small tower of blocks. During the learning phase it observed the instructor interacting with the blocks. The demonstration involved constructing towers, knocking them down and then rebuilding them (some-

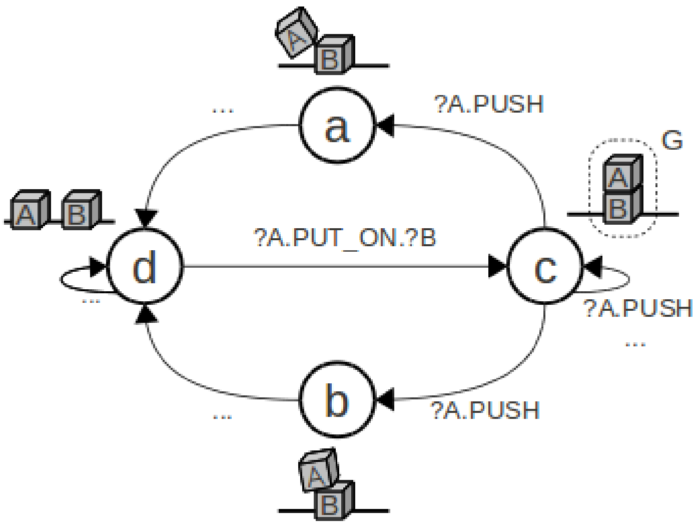


Figure 6.1: Behaviour Graph of a Generalised 2 Block System. The system has four states: **a**. block ?A is falling; **b**. block ?A is partially on block ?B but teetering on the edge; **c**. block ?A is on block ?B, forming a group ?G; **d**. blocks ?A and ?B are on the floor.

times with different objects and sometimes with the same objects). The agent created several histories of constructed towers from the observations.

Using the histories generated from observation snapshots the agent was also able to create generalised models of the behaviour of small towers that were 2 or 3 blocks high. Figure 6.1 shows an example generalised model produced by the agent describing the process of constructing a 2 block tower where the agent can put one block on another and can push it off. The transitions of the model are shown in table 6.1.

The generalised model uses appropriate variables: specific block identifiers have been replaced with variables; the floor object has not been replaced by a variable because the agent only ever observed the system in action while on the floor. The model contains no anomalous states or transitions that are obviously incorrect. It has also included two different ways in which the top block can fall to the floor once it has been pushed (either falling straight to the floor or teetering on the edge before falling to the floor). Note that not all possible transitions have been observed - the agent has not seen the teetering block actually falling and therefore a transition from state **b** to **a** is missing. This transition could be added after further observations.

	transition		**deltas**	
a →	...	→ d		
b →	...	→ d		
c →	...	→ c		
d →	...	→ d		
c →	?A.PUSH	→ c		
c →	?A.PUSH	→ a	?G.height	-
c →	?A.PUSH	→ b		
d →	?A.PUT_ON.?B	→ c		

Table 6.1: Transitions of a Generalised 2 Block System. (?G is an identifier for the group object formed by blocks ?A and ?B)

The agent also learned similar but more complex models for towers with more blocks, an example is shown in figure 6.2. These were used as the agent's knowledge-base when it was given the task of constructing a tower using its planning algorithm.

The agent was able to successfully build small towers of blocks by finding a good match from the systems it had learned. For each task, the agent was able to find a system with a context that was applicable to its current world state *and* also had a state which satisfied the goal state.

The agent was able to recover if something went wrong during the execution of a plan. Occasionally an action would not have its intended effect (a block is not put in the correct place). Also the instructor would sometimes execute actions which undid the agent's actions. In these cases the agent was able to continue acting using its original plan (albeit from an unexpected state), or if the system was no longer applicable, then the agent

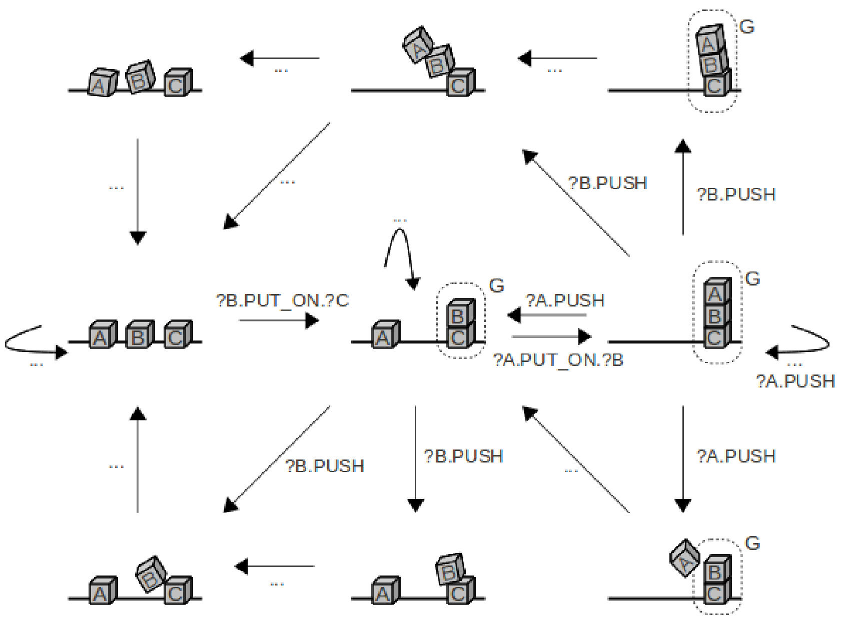


Figure 6.2: Behaviour Graph of a Generalised 3 Block System.

was able to construct an entirely new plan and proceed from there. In all cases, the persistent nature of the planning algorithm always ensured that the tower was eventually constructed.

6.5 Limitations of the Agent

The matching, model refinement and planning algorithms presented in this chapter have significant limitations. Their primary purpose is demonstrate that the Q-System representation is suitable for use by an autonomous agent in the complex tasks of learning and planning. Their secondary purpose is to explore the types of problems that will be encountered when developing a more advanced agent. This section describes the most significant limitations that would need to be addressed in order to create an agent that could operate effectively and robustly in a complex environment.

The Matcher

The matcher is currently limited in the extent to which object relationships are inspected when scoring the similarity of objects. The matcher only checks that objects have similar relationship types (e.g. they both have an 'on' relationship with some other object). It does not measure the similarity of the object on the end of the relationship or any properties of the relationship itself. This will cause the matcher to perform badly in certain situations where properties of related objects are important in defining the role of the object.

When matching entire systems the matcher requires that states must be identical to be matched. This approach is problematic where the scope of the two systems being matched is significantly different. If one system describes a sub-system of a larger system then it may describe fewer objects in its states than a model of the complete system. However the states in the smaller system should still be able to match to those of the larger system (they are subsets of the larger systems states).

The Learner

A major limitation of the learner is that its parameters are set arbitrarily. These include the generalisation threshold and also the weights used to score system matchings. These parameters are critical in shaping the types of models that the learner produces. An analysis of the effect of these parameters on models is required to understand how they affect learning. Ideally they be would assessed using task performance of the agent over a variety of different tasks. However this approach is difficult because the overall task performance of the agent is dependent on the integrated result of several complex systems (i.e. the vision, history generating and planning system), not just the learner itself.

Another issue with the learner is that it has no way of removing spurious actions and states that have been recorded by the history generator. The learner includes *all* observed transitions in the combined models. Transitions that ideally shouldn't be in the system (such as the effects of another agent's actions) should be removed. If the agent has access to a large number of observations, then this could be achieved by implementing a statistical significance test that would remove any transitions that fall below a certain threshold of significance. This would solve the problem but would have to be done carefully to avoid removing rarely seen transitions which should be part of the observed system.

The Planner

The main limitation of the planner is that it can only plan using a single system. Therefore to achieve a goal the agent must have a system in its knowledge-base that includes both its current state and the goal state. The planner could achieve many more tasks if it were able to plan across multiple systems. This could be achieved by coupling together multiple systems. Systems would be coupled by finding compatible states and allowing world objects to play roles in both systems simultaneously. This would allow the planner to work across multiple coupled systems by 'exiting' one system and 'entering' another at compatible states.

Another limitation of the planner is the way in which it handles ambiguous action effects by always expecting the most likely outcome. This approach is problematic when the most likely outcome is a *self-loop* (in which the action returns the system to the current state). In these cases the planner should recognise that repeating the action will normally cause the system to transition to a new state eventually.[3] Also the planner can fail if

[3]This an area that could also be improved in the Q-System representation itself. Systems could distinguish between self-loop actions that *always* eventually exit and those that sometimes do not. In fact, this technique is just one of a number of ideas that have been investigated in a recent research project [3]. It focused on how Q-Systems can be used for planning in qualitative domains. It found that the limitations described in this section can be overcome with some small extensions to the Q-System representation.

an ambiguous action leads to an unexpected state in which the goal cannot be reached. Ideally the planner would distinguish between safe and unsafe routes through the system. A 'safe' route would avoid any actions in which the agent could end up in dead end state.

The planner is also limited to relying on a naive search when matching systems to the world state. This will not work if there are a large number of objects in the world to match against. A better approach is to begin by matching objects from the current goal and then to search for objects based on their context in the candidate system.

(Note, the honours project was focused purely on planning whereas the work in this thesis focused on development of a *learnable* representation.)

Chapter 7

Conclusion

Overall the project has achieved its goals of creating an integrated action and qualitative process representation, constructing a rich simulation for the purpose of testing the representation on real problems, and evaluating the representation in terms of its suitability for use in learning and planning in complex worlds.

This chapter summarises these contributions. It also draws together several insights regarding the issues of creating and learning the representation that were discovered during the project. The chapter finishes with a discussion of possible next steps in the development and evaluation of an integrated learning system.

Representation

Q-Systems (chapter 3) is a novel symbolic level representation which integrates two important existing representational frameworks: action description languages (section 2.1) and qualitative process modelling (section 2.3). It allows world dynamics to be modelled in a new way that enhances the ability of an agent to learn and plan.

One way in which Q-Systems extends action description languages is that instead of just focusing on the effects of individual actions, Q-Systems allows multiple actions to be combined into an encapsulated subset of world behaviour. This encapsulation allows an agent to focus on small numbers of interrelated objects when learning. This modular approach is similar to the concept of qualitative process models [14], however a Q-System model explicitly enumerates the states and transitions in the system whereas qualitative models infer the state from qualitative relationships. Also, actions are not represented in qualitative process models.

All the actions and transitions in a Q-System share the same context which acts as a pre-condition. This means that when an action is being learned, it need only be learned in its current context. Other action learning systems [35][40][45] attempt to learn a complete model of an action, i.e. the effects of an action in all possible situations in which it might be used. This makes it very difficult to learn a correct model for a generic action because there are so many different situations in which it can be applied. This is in contrast to Q-Systems in which the effects of an action need only be specified for very specific situations.

Because Q-Systems integrates both actions and qualitative behaviour it is possible to learn both simultaneously, which means learning can be achieved in a single process instead of two independent processes. Action learning systems typically focus on actions only and treat world dynamics as a separate problem. This divided learning approach is an un-natural way of tackling the problem, because actions and world dynamics can both be considered instances of the same thing: 'an event which changes the current state of the world'. The Q-System representation models them in this way and this allows a learner to take advantage of the similarities between the two types of transition (action or world dynamic) when constructing models.

Uncertainty in action outcomes has to be represented if models are to be useful in realistic environments. Q-Systems include a simple way of modelling probabilities of action outcomes. A state with many outgoing transitions all using the same action would be very difficult to plan with without knowing which outcomes are more likely. Q-Systems uses a frequency count on each transition to indicate the likelihood of a particular transition occurring. This is similar to methods used in other action description languages [31][45]. Q-Systems also allow for the introduction of hidden variables which can be used to disambiguate states in which the action outcome depends on the hidden variable.

Q-Systems introduce the notion of adding 'qualitative deltas' to action models. Qualitative deltas help to bridge the gap between the qualitative and quantitative worlds without having to resort to fully quantitative modelling. A Q-Systems qualitative delta represents an observation that a measure has changed (increased or decreased) even though the *qualitative* value of the measure is still the same. This information is vital to modelling certain processes because actions may have no observable effect other than qualitative deltas. If deltas were not modelled then such actions would be incorrectly learned as having no effects (an example is the action of reducing the flow from a tap, the qualitative state of the world has not changed but there was an effect). No other action description language uses qualitative deltas.

A goal of this project was to determine if a purely qualitative approach was sufficient in realistic environments. Q-Systems have shown that using only qualitative information in action models allows many interesting mechanisms to be modelled. However, there are some situations in which at least an estimate of quantitative information is required. For example, states within the sink system (see section 4.5.1) cannot distinguish between a sink that is filling with a dribble of water and a sink that is filling with a reasonable amount. In this situation, having quantitative information about the flow rate would enable the agent to estimate how long the sink will take to fill and therefore take action to speed up the process if required. A possible solution to this problem is discussed in the section on future work.

Simulation

A simulation framework and two richly simulated interactive worlds were created for this work. The simulated worlds provided realistic qualitative observations of complex behaviour in real-time. These were used to develop the Q-System representation and learning system.

The simulation framework was built using a commercial physics game engine that generated the complex physical interactions between objects. This type of framework has been used in symbolic level agent research before [45], however, the framework presented in this work (see chapter 4) implements a number of features that have not been used in previous work on action learning systems. These include: observation of qualitative variables and shape descriptions, simulation and observation of liquids, and observation of 'group' objects. This richness of the state description allows more interesting behaviours to be explored and learned. Overall, the simulation adequately generated the types of complex behaviours that the agent was required to learn about. It was able to simulate a range of real world complexities such as hidden state, unbounded numbers of objects, multiple actors, etc.

The simulation's vision system had to be carefully constructed to ensure that the observations it generated were consistent with what a real vision might be plausibly expected to produce. It was found that subtle interactions between the timing of updates within the simulation and vision system could create artifacts that would not be expected in a 'perfect' vision system. Such timing issues could cause the vision system to miss a qualitative transition or incorrectly attribute effects to an action (these are explained in section 4.3). But these types of mistakes are the kind that would be expected in a real 'imperfect' vision system. Therefore they were not removed from observation snapshots, but were passed on to the learning system which had to account for them when creating models.

A key part of the vision system was observations of qualitative shape and space. In fact these were critical to the types of behaviour that could be learned about. For example, learning about a see-saw mechanism was impossible unless the agent could observe the difference between an object simply being 'on' another and an object that is on another at a specific location (e.g. the end, or the middle).

The vision system uses a basic set of qualitative shape and space descriptors which allows the agent to learn a number of interesting mechanisms. However, ideally it would use a more powerful set of descriptors. This would allow a wider range of mechanisms to be modelled at a level of detail sufficient for predicting their important behaviour. What a sufficiently powerful representation of shape and space would look like is an open research question [8].

The simulation, vision and skill systems will be made available electronically for other researchers to use.

Histories

This thesis proposes a two stage learning mechanism. The first stage is to extract 'histories' from real-time observations - short sequences of context restricted observations. The second stage is to construct generalized models from the histories.

The use of histories is a departure from other action learning systems which learn from observations of *individual* transitions rather than from sequences of transitions with a common context, as is the case with histories. This is an important difference because it makes learning easier in two ways. Firstly, histories allow the *cause* of hidden variables to be identified from previous actions. In the case of learning from individual transitions, hidden variables are problematic because they manifest themselves as two (or more) examples in which an identical state and action lead to two *different* states. By using histories the cause of this ambiguity can potentially be resolved by examining the transitions which lead up to the before state. If it is the case that particular prior actions *always* cause the ambiguous action to have a certain outcome, then this can be exploited by a system which learns from histories.

The second benefit of learning from histories is that they reduce the size of the states that are used by the generaliser. Histories are constructed over a duration of several snapshots and are therefore able to identify relevant objects and remove unrelated observations. Other action learning systems which do not filter in this way must limit the size of their state descriptions. This is often achieved by reducing the number of objects in the world to an unrealistically small amount.

It was shown that with appropriate heuristics for controlling the contextual and temporal scope, histories can be automatically generated that are appropriate for learning subsets of world behaviour as Q-Systems. The types of histories generated were instances of the types of models that would be expected to be useful for planning.

Two types of heuristics were experimented with: temporal and contextual. The temporal heuristics were used to control the points at which histories started and stopped. They used mechanisms such as identifying interesting events, counting actions elapsed, or counting system cycles to trigger the start or end of a particular history. Using a simple mechanism such as counting the number of actions elapsed was sufficient to create isolated pieces of behaviour that could be used to generate a full system model if several histories were observed and combined. However, more sophisticated heuristics will be required to generate histories that are *likely* to contain most states and transitions of a target system. Histories with a more complete description will improve learning since fewer histories will be required to complete a model. How significant an improvement this might be is an open research question.

The contextual heuristics were based on identifying objects that change and their relationship with objects already being tracked. Using a naive approach of including *any* object which changes was sufficient in a restricted environment. This approach will break down in a realistic environment with lots of simultaneously changing objects (caused by other agents and processes). To address this problem a 'relatedness' heuristic was introduced which estimated how likely it is that a changing object is actually part of the currently observed system.

The history generator was evaluated by generating lots of histories and looking at statistics such as the average number of states, context objects, etc. Also the generated histories were compared with the types of systems that would be expected to be useful in the particular domain. This was sufficient to show that the histories being generated were appropriate for learning. However it could not be determined which combinations of heuristics were better than others for use in learning. This could be achieved by using the histories as input to a learner, and then assessing the resulting models with respect to task achievement.

Learning & Planning

A complete agent that can learn generalised models and use them to plan and achieve goals was implemented and tested using the simulation. It demonstrated that Q-Systems can be used successfully in generalisation and planning. However, the algorithms were somewhat limited in the types of models that can be constructed and also in the sophistication of the resulting plans. Improving the performance of these algorithms is an area of future work (some preliminary ideas are discussed in section 7.1). This section describes some of the issues encountered during development of the learner and planner.

Matching was identified as a critical component in generalising. Generalizing with Q-Systems requires entire system matching, which involves systems' contexts, states, and transitions. The problem of searching the exponential search space of possible matchings was addressed and a polynomial time 'good enough' algorithm (see section 6.1.1) was developed and applied with promising results.

Q-Systems make planning easier by constraining the search space of possible actions. Planning with a single system was found to be mostly trivial (except for the problem of resolving ambiguous actions). The planner worked by constructing a subset of an applicable system with the property that the goal can be reached from all states. It then performed a straightforward shortest path search to the goal.

A more difficult problem was that of finding a system that is applicable to the agent's current situation (i.e. matching abstract generalized models to concrete situations). A naive search was used by the agent presented in section 6.3. However, because this is a type of matching problem a more efficient solution could be developed by re-using the ideas developed for the learner's matcher (described in section 6.1).

A problem that is particular to planning with Q-Systems is that achieving more complex goals will require the ability to chain multiple systems together. The obvious solution is to bind objects in different systems and then to plan across the systems as though each system were a large composite action (similar to Erol et al.[11]). How this mechanism would work in practice needs to be further explored.

Q-Systems have been shown to work with an integrated history generating, learning and planning agent in a complex world. However a limitation of this thesis is it has not shown in a quantitative experiment that this approach is better at solving particular tasks or dealing with certain environmental complexities when compared with existing action representations. It is clear that the representation can express more than what an action representation or qualitative process representation alone can express, however a task orientated experiment comparing the approaches side-by-side would help identify the particular strengths and weaknesses that cannot be seen when assessing the representation in isolation.

7.1 Future Work

There are three related areas of work that would benefit from further investigation: improving the representation, improving the integrated learning system, and finally, development of better means to evaluate different representations and learning systems.

The representation could be improved by adding extra information to the system models. One kind of extra information, 'entry' and 'exit' actions, relates to how Q-Systems are used in planning. Adding additional annotated actions of this type would make planning across multiple systems easier. The actions would represent the actions that normally make the system applicable or not applicable (i.e. they complete or invalidate a system's context). This information could be obtained by observing the action that enter or exit a system and finding which ones happen most frequently. Planning across multiple systems would be made easier because the entry and exit actions can be prioritized when searching for actions that join systems.

A second improvement to the representation would be to introduce a limited amount of quantitative information. Q-Systems were designed to make full use of qualitative information especially in situations where quantitative information is unavailable. However, quantitative information will be sometimes available; extending Q-Systems with quantitative information would allow an agent to take full advantage of all its observational capabilities.

One way to introduce quantitative information into systems would be to annotate qualitative deltas with an estimated amount of change. This would solve the problem in which there is sometimes no observable qualitative difference between deltas which are in fact significantly different. For example, the qualitative observation of turning a lever by a small amount is identical[1] to the observation of turning a lever by a large amount. The difference would be captured by the representation if the action's qualitative delta had a quantitative estimate of how far the tap was turned.

[1]Identical if we assume that no natural comparison points exist and that arbitrary landmarks have not been set.

A complication of introducing quantitative estimates is that actions may need to be quantitatively parameterized in order to allow the quantitative estimates to be used effectively in reasoning. For example, the 'turn_lever' action would need to become 'turn_lever_X_degrees'. This approach would allow an agent to model behaviour more precisely than with purely qualitative information.

The second significant area of future work is to improve the overall learning system of the agent such that it is still effective when additional complexities are introduced into the environment. Two particular issues are how to learn about hidden variables and also how to learn when multiple other agents and processes are present. Q-Systems support the introduction of hidden variables into states, but the current learning system does not take advantage of this. A mechanism for identifying hidden variables could make use of the temporal aspect of example histories. If certain ambiguous actions outcomes are always preceded by the same actions then the histories will show this. Correctly modelling hidden variables would improve models and the plans that are generated from them.

Learning can also be improved by filtering out the changes to the environment that are unrelated to the agent's current focus. These changes arise from other agents' actions and also other naturally occurring processes. The current heuristics used in the history generator assume that no other agents or processes are present. Ideally the 'relatedness' measure would ensure that unrelated objects are not included in the current history. The best way to achieve this is not obvious.

The final area of future work is development of a robust means of comparing different representations and learning systems against fixed benchmarks.

The learning system presented in this thesis is composed of several integrated sub-systems. The performance of each sub-system is dependent on a particular configuration of heuristics, parameter settings and other design decisions. Assessing the effect of changing a *single* design decision (swapping one heuristic for another for example) must be measured by the performance of the learning system as a whole. A good way to assess overall learning performance is to use benchmark tasks which reflect the types of tasks the agent should be able to achieve. One such task is the building towers task (see section 6.4). This is a good benchmark because performance can be unambiguously assessed (e.g. what is the tallest tower that was built) and also because it involves complex qualitative behavioural dynamics. Future work would focus on measuring the effect of various heuristics and other design decisions on the ability of the agent to perform well on this and other benchmarks.

The benchmarks would also be used to compare Q-Systems with existing representations and learning systems. Ideally these other systems would already be documented using comparable benchmarks; however this is not the case. Some systems use superficially similar scenarios, e.g. the tower building task, but the environments in which the task is executed are significantly different and therefore a direct comparison is not possible. To make direct comparison possible the existing systems would have to be reimplemented and applied to identical tasks in the same environment.

Bibliography

[1] ALLEN, J. F. Maintaining knowledge about temporal intervals. *Commun. ACM 26*, 11 (November 1983), 832–843.

[2] AMIR, E. Learning partially observable action models, 2005.

[3] BEBBINGTON, J. Multi system qualitative planner. Tech. rep., Engineering and Computer Science, Victoria University of Wellington, 2010.

[4] BLOCKEEL, H., AND RAEDT, L. D. Top-down induction of first-order logical decision trees. *Artificial Intelligence 101*, 1-2 (1998), 285 – 297.

[5] BOXER, P. A. Towards learning naive physics by visual observation: Qualitative spatial representations. In *AI '01: Proceedings of the 14th Australian Joint Conference on Artificial Intelligence* (London, UK, 2001), Springer-Verlag, pp. 62–70.

[6] BRATKO, I., AND ŠUC, D. Learning qualitative models. *AI Mag. 24*, 4 (2004), 107–119.

[7] BRUCE, P. C., PORTER, B., AND BATORY, D. A compositional approach to representing planning operators. Tech. rep., University of Texas, 1996.

[8] COHN, A. G., AND HAZARIKA, S. M. Qualitative spatial representation and reasoning: An overview. *Fundam. Inform. 46*, 1-2 (2001), 1–29.

[9] DAVIS, E. Pouring liquids: A study in commonsense physical reasoning. *Artificial Intelligence 172*, 12-13 (2008), 1540 – 1578.

[10] DE KLEER, J. Qualitative and quantitative knowledge in classical mechanics, technical report 352. Tech. rep., MIT AI Lab, 1975.

[11] EROL, K., HENDLER, J., AND NAU, D. S. Htn planning: Complexity and expressivity. In *In AAAI-94* (1994).

[12] ESPOSITO, F., SEMERARO, G., FANIZZI, N., AND FERILLI, S. Conceptual change in learning naive physics: The computational model as a theory revision process. In *AI*IA* (2000), vol. 1792 of *Lecture Notes in Computer Science*, Springer, pp. 214–225.

[13] FIKES, R., AND NILSSON, N. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2* (1971), 189–208.

[14] FORBUS, K. D. Qualitative process theory. *Artificial Intelligence 24*, 1-3 (1984), 85 – 168.

[15] FORBUS, K. D. Introducing actions into qualitative simulation. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (San Mateo, California, 1989), N. S. Sridharan, Ed., Morgan Kaufmann, pp. 1273–1278.

[16] FORBUS, K. D., NIELSEN, P., AND FALTINGS, B. Qualitative kinematics: a framework. In *IJCAI'87: Proceedings of the 10th international joint conference on Artificial intelligence* (San Francisco, CA, USA, 1987), Morgan Kaufmann Publishers Inc., pp. 430–435.

[17] FOX, M., AND LONG, D. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of AI Research. 20* (2003).

[18] GIL, Y. Learning by experimentation: incremental refinement of incomplete planning domains. *Proceedings of the Eleventh International Conference on Machine Learning* (1994).

[19] GUESGEN, H. W. Spatial reasoning based on allen's temporal logic. Tech. rep., International Computer Science Institute, 1989.

[20] HAYES, P. J. *The second naive physics manifesto*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990, pp. 46–63.

[21] HOGGE, J. C. Compiling plan operators from domains expressed in qualitative process theory. In *AAAI* (1987), pp. 229–233.

[22] KUIPERS, B. *Qualitative reasoning: modeling and simulation with incomplete knowledge*. Massachusetts Institute of Technology, Cambridge, Mass., USA, 1994, ch. 2.

[23] KUIPERS, B. J. Qualitative simulation. *Artificial Intelligence 29* (1986), 289–338.

[24] LI, R., AND PEREIRA, L. M. Representing and reasoning about concurrent actions with abductive logic programs. *Annals of Mathematics and Artificial Intelligence 21*, 2-4 (1997), 245–303.

[25] LIFSCHITZ, V., AND REN, W. A modular action description language. In *AAAI'06: Proceedings of the 21st national conference on Artificial intelligence* (2006), AAAI Press, pp. 853–859.

[26] MCCARTHY, J., AND HAYES, P. J. Some philosophical problems from the standpoint of artificial intelligence (1969). - (1987), 26–45.

[27] MCDERMOTT, D. Pddl — the planning domain definition language, 1998.

[28] MITCHELL, T. M. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1997, ch. 2.

[29] MUGAN, J., AND KUIPERS, B. Learning to predict the effects of actions: synergy between rules and landmarks. In *IEEE International Conference on Development and Learning (ICDL-07).* (2007).

[30] MUGGLETON, S. Inductive Logic Programming. *New Generation Computing 8*, 4 (1991), 295–318.

[31] OATES, T., AND COHEN, P. R. Searching for planning operators with context-dependent and probabilistic effects. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference, Vol. 2* (Menlo Park, California, 1996), H. Shrobe and T. Senator, Eds., AAAI Press, pp. 865–868.

[32] PEDNAULT, E. P. D. Adl: exploring the middle ground between strips and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning* (San Francisco, CA, USA, 1989), Morgan Kaufmann Publishers Inc., pp. 324–332.

[33] PEDNAULT, E. P. D. Adl and the state-transition model of action. *Journal of Logic and Computation 4*, 5 (1994), 467–512.

[34] REITER, R. A logic for default reasoning. *Artificial Intelligence 13*, 1-2 (1980), 81 – 132.

[35] SHAHAF, D. Learning partially obserevable action schemas. In *Proceedings of the Twenty-First AAAI Conference on Artificial Intelligence* (2006).

[36] SHEN, W.-M. Discovery as autonomous learning from the environment. *Machine Learning 12* (1993), 143–165.

[37] STEINHAUER, H. J. *A Representation Scheme for Description and Reconstruction of Object Configurations Based on Qualitative Relations*. PhD

thesis, Linkping UniversityLinkping University, CASL - Cognitive Autonomous Systems Laboratory, The Institute of Technology, 2008.

[38] THOMAZ, A. L., AND BREAZEAL, C. Teachable robots: Understanding human teaching behavior to build more effective robot learners. *Artif. Intell. 172*, 6-7 (2008), 716–737.

[39] VELOSO, M., CARBONELL, J., PREZ, A., BORRAJO, D., FINK, E., AND BLYTHE, J. Integrating planning and learning: The prodigy architecture. *Journal of Experimental and Theoretical Artificial Intelligence 7* (1995), 81–120.

[40] WANG, X., SIMON, H. A., LEHMAN, J. F., AND FISHER, D. H. Learning planning operators by observation and practice. In *Proceedings of the Second International Conference on AI Planning Systems, AIPS-94* (1996), pp. 335–340.

[41] WATKINS, C. J. C. H., AND DAYAN, P. Technical note: Q-learning. *Mach. Learn. 8*, 3-4 (1992), 279–292.

[42] WINOGRAD, T. Procedures as a representation for data in a computer program for understanding natural language. Tech. rep., MIT AI Lab, MAC-TR-84, 1971.

[43] WINSTON, P. H. Learning structural descriptions from examples. Tech. rep., MIT, Cambridge, MA, USA, 1970.

[44] YANG, Q., WU, K., AND JIANG, Y. Learning action models from plan examples with incomplete knowledge. In *Proceedings of the 2005 International Conference on Automated Planning and Scheduling, (ICAPS 2005) Monterey, CA USA* (2005), pp. 241–250.

[45] ZETTLEMOYER, L. S., PASULA, H., AND KAELBLING, L. P. Learning planning rules in noisy stochastic worlds. In *AAAI* (2005), pp. 911–918.