Inductive learning of knowledge-based planning operators

Inductive Learning of Knowledge-Based Planning Operators

Inductive Learning of Knowledge-Based Planning Operators

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Rijksuniversiteit Limburg te Maastricht,
op gezag van de Rector Magnificus, Prof. mr. M.J. Cohen,
volgens het besluit van het College van Dekanen,
in het openbaar te verdedigen
op donderdag 7 maart 1996 om 14.00 uur

door

Timothy John Grant

Promotores: Prof. dr. H.J. van den Herik
Prof. dr. P.T.W. Hudson

Leden van de beoordelingscommissie:
Prof. dr. ir. A. Hasman (voorzitter)
Prof. dr. ir. J.L.G. Dietz (Technische Universiteit Delft)
Dr J.C. Hage
Prof. dr. L. Steels (Vrije Universiteit Brussel, België)
Prof. dr. A. Tate (University of Edinburgh, Scotland)

Preface

This thesis shows that behavioural domain knowledge (planning operators) can be induced from structural knowledge of the domain. Example domains have been drawn from my personal experience, as well as from the AI literature, with space applications being particularly important.

My research has been made possible by many people. I acknowledge the key contributions, in chronological order, of:

- Alan Bond, the inventor of the HOTOL engine, who triggered my interests in research and AI by persuading me to analyse the computer systems for a starship (Grant, 1978);
- my family, who gave me time and space for research despite neither understanding my topic nor seeing any benefit in it;
- the members of the UK Planning and Scheduling Special Interest Group, who gave me stimulation and companionship for more than a decade;
- Joop Renes, who gave me a job and encouraged me to register for a PhD in The Netherlands;
- Daan de Hoop, Netherlands Agency for Aerospace Programs, who sponsored the use of a barely-emerging information technology for space applications;
- past and present Space Group colleagues in Origin/BSO, whose *esprit de corps* made it a pleasure to work together on challenging information technology applications;
- Mark Wigmans, whose chance remark while developing the High Performance Capillary Electrophoresis Payload Simulator (Grant, Wigmans, Eckhard, van Eenennaam, 1992) led me to the idea of partitioning version spaces by object-classes and -instances;
- Padre Antonio Soler, Heinrich Ignaz Franz von Biber, Claudio Monteverdi, and Henry Purcell, who kept me going during the writing phase;
- Marc Heppener, Space Research Organisation Netherlands, who convinced me that there were real applications for my research, and who pointed me towards the right promotors;
- Jaap van den Herik and Patrick Hudson, who guided my research to its conclusion, in a majestic process akin to docking a Space Shuttle to the International Space Station.

I dedicate this thesis to the anonymous referee who rejected a paper I had submitted to a prestigious AI conference with the words:

> "The idea seems to be that You want to find the planning operators that can be used when the planner encounters a new domain. This seems like a very bad idea to me."

Contents

Figures

1 INTRODUCTION

1.1 MOTIVATION

1.1.1 Research Questions

In the Artificial Intelligence (AI) research field of Knowledge-Based Planning, action in the world is often represented in the form of planning operators. This thesis addresses the research question:

"Where do planning operators come from?"

Planning operators are data-structures used in generating plans, as production rules are used in expert systems. Figure 1 shows an example planning operator, as represented in the STanford Research Institute Planning System (STRIPS) formalism (Fikes and Nilsson, 1971). The operator represents some knowledge about a class of possible actions in a problem domain. An example problem domain is the *blocks world*, first described by Winograd (1972) and used by many authors in planning research. In the blocks world, robot hands manipulate blocks stacked on tables. The planning operators for the blocks world - of which **`stack`** is one - are used to generate plans, i.e., sequences of manipulations intended to change an initial configuration of blocks into a desired, goal configuration. The **`stack`** operator can be read as follows: "The preconditions for stacking block x on block y are that the (robot) hand is holding block x and that the top of block y is clear. When the operator is executed, these preconditions are deleted from the modelled world-state and three postconditions are added, namely that the hand is empty, block x is on block y, and the top of block x is clear."

stack(x,y)

Preconditions:	*HOLDING(x), CLEAR(y)*
Delete-list:	*HOLDING(x), CLEAR(y)*
Add-list:	*HANDEMPTY, ON(x,y), CLEAR(x)*

Figure 1: Example Planning Operator, from Nilsson (1980).

Plan generation - also known as *plan construction* (Allen, Kautz, Pelavin and Tenenberg, 1991), *plan synthesis* (Georgeff, 1987), or, more simply, as *planning* (Tate, Hendler and Drummond, 1990) - is defined as the process of selecting and instantiating actions from a set of planning operators and logically ordering them in a sequence that will, on execution, transform a given initial domain state into a desired ("goal") domain state. It takes a set of planning operators, an initial state description, and a goal state description as inputs, and outputs a *plan*, i.e., a sequence of instantiated operators. For this reason, plan generation is also known in Operations Research as *sequencing*. The initial and goal state-description pair may be termed the *planning problem*. Plan generation (including the STRIPS formalism) is reviewed in Chapter 2, and the blocks world is detailed in Chapter 4.

Figure 2 shows a typical plan for a blocks world in which there are two blocks, one hand, and one table. The plan is depicted in the form of states and transitions, with State1 as the initial state and State3 as the goal state. The states are described using Nilsson's (1980) descriptions for blocks world states, as well as being shown pictorially. The transitions are shown as labelled arrows. The plan is the sequence: [**`pickup b1`**] followed by [**`stack b1 on b2`**].

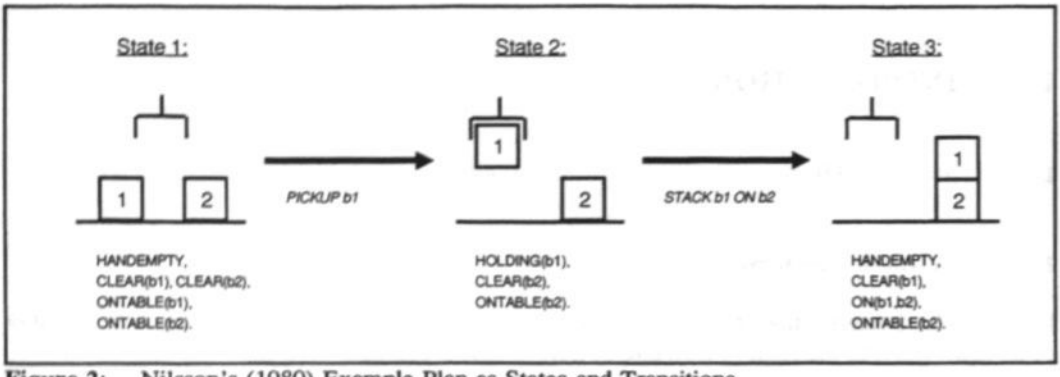

Figure 2: Nilsson's (1980) Example Plan as States and Transitions.

At a superficial level, my research question is quickly answered: the planning operators for the Monkey and Bananas Problem (Köhler, 1917) come from Fikes and Nilsson (1971), those for the blocks world come from Nilsson (1980), those for house-building come from Tate (1975), those for fighting forest fires come from Cohen, Greenberg, Hart and Howe (1989), and so on. Fikes, Nilsson, Tate and Cohen *et al.* were the developers of the planning systems involved. Hence, it appears that planning-system developers are responsible for providing suitable planning operators, in the same way as knowledge engineers are responsible for providing suitable production rules in expert systems.

This answer immediately begs further questions, such as:

- How does a developer formulate a set of planning operators for a new domain? Is this difficult? Is this a transferable skill?

- How can the developers (and the users) be sure that the set of planning operators is complete, correct and precise[1]?

- What are the consequences of having incomplete, incorrect, or imprecise domain knowledge? In particular, what happens if the domain knowledge is distributed over multiple agents?

In expert systems, there has been progress in automating the formulation of production rules, especially in inducing rules from examples. This suggests that it might be possible to induce planning operators from examples. This possibility motivated my research.

1.1.2 Thesis Claim

I claim that STRIPS-style planning operators can be induced *ab initio* from a domain model represented as unordered lists of domain objects, inter-object relationships, and interrelationship constraints. Moreover, such a domain model can be extracted from an unordered list of state-descriptions.

I employed a hybrid methodology (Cohen, 1991) in my research. My claim was substantiated by developing an induction algorithm (known as the *Planning Operator Induction* (POI) algorithm), by implementing the algorithm, and by performing various experiments on the implemented software. To

[1] The intuitive meanings of the terms are intended here.

do this, I have had to develop:

- *the POI ontology*. An *ontology* is a set of definitions of content-specific knowledge-representation primitives that is both human and machine readable (Gruber, 1992). Example primitives are classes, relations, functions, and object constants. The POI ontology is based on primitives found in the *entity-relationship model* (Chen, 1976) of domain structure and in the *state-transition model* of domain change. The POI ontology is defined in Chapter 3.

- *a family of meta-heuristics*. A *meta-heuristic* is a form of control knowledge that applies to a set of domains (Sowa, 1984), i.e., it is domain-independent. The meta-heuristics are used within the POI algorithm for identifying valid transitions between domain states. More details are given in Chapter 3.

The research reported in this thesis differs from previous approaches to learning planning knowledge in that it:

- *induces domain knowledge. Domain knowledge* "describes the world and the actions that are available to the planner" (Minton and Zweben 1993, p. 2). *Control knowledge* indicates how the planner should control its search for a plan. Minton and Zweben observe that "most of the research to date has concentrated on learning control knowledge ..." (*ibid.*, p. 14).

- *induces behavioural knowledge from structural knowledge*. Minton and Zweben's (1993) definition implies that there are two subclasses of domain knowledge: knowledge about the structure of the domain (*structural knowledge*), and knowledge about the available actions in that domain (*behavioural knowledge*). Previous approaches have been restricted to behavioural knowledge.

- *represents the induced knowledge as planning operators*. A variety of knowledge representations are used in other induction systems. For example, in rule induction the induced knowledge is represented as production rules or decision trees.

- *models domains using an ontology grounded on mature software engineering principles*. Other researchers have largely failed to document the ontology they used for representing domain knowledge.

- *enables the induction of behavioural domain knowledge ab initio*. My research has not been concerned with automating knowledge refinement, but rather with "the far more difficult and seldomly encountered phenomenon of formulating radically new theories from ground zero" (Carbonell and Gil, 1990, p. 193).

- *does not make use of any control knowledge*. Other algorithms for learning domain knowledge for planning purposes depend, wholly or partly, on inputs that provide domain-specific information on the sequencing of states and/or actions, i.e., control knowledge.

1.1.3 My Research in Context

Knowledge-based planning research has passed through three eras. Each era is associated with a set of topics which were at the "leading edge" when the era opened. Research continues in all topics. In the first era (Figure 3), research focused on inventing and refining algorithms for generating plans. These algorithms input operators, planning problems, and control knowledge, and output plans.

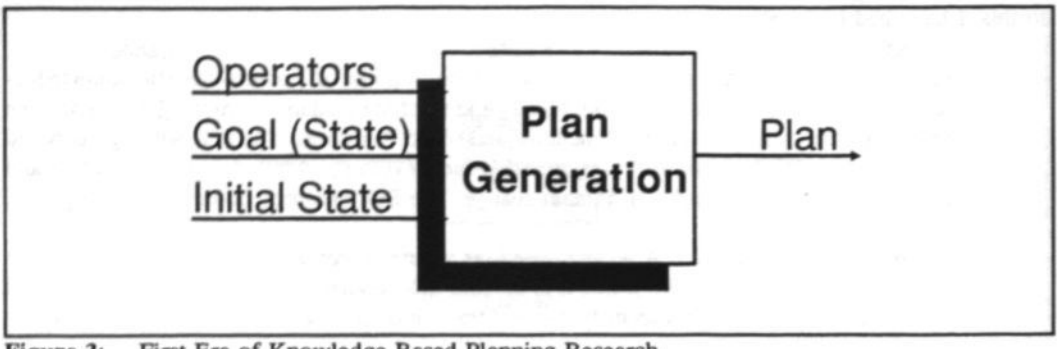

Figure 3: First Era of Knowledge-Based Planning Research.

The second era opened in the mid-1980s when the focus of knowledge-based planning research shifted to scheduling, to executing the generated schedules, to monitoring for schedule failure, to recovering from such failures (Ambros-Ingerson and Steel, 1988), to analysing plans, and to recognising intentions in plans. The second era was concerned with processes that are "downstream" of plan generation (see Figure 4), because they take the output of plan generation as their input. The Control process combines plan execution, monitoring, failure detection, and recovery.

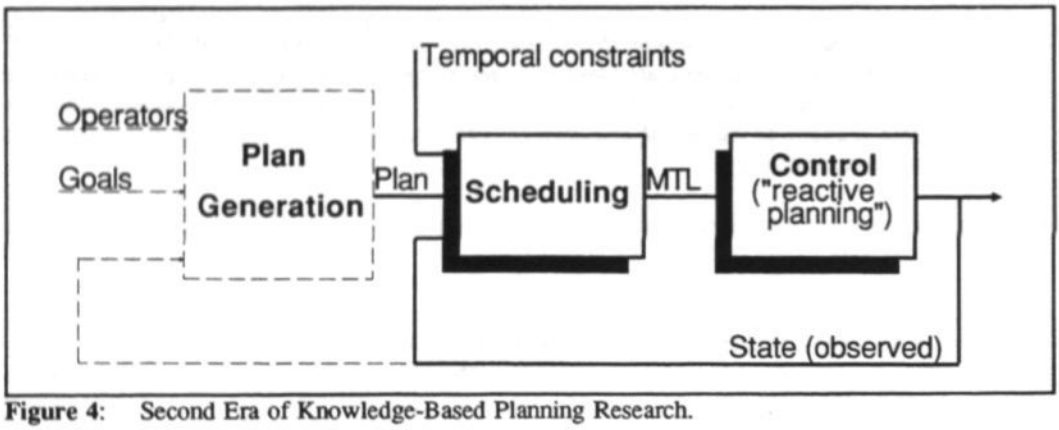

Figure 4: Second Era of Knowledge-Based Planning Research.

Until recently, a clear distinction has been made between plan generation and scheduling. Steel (1988, p. 146) states that: "scheduling takes a partially ordered set of actions as a given, and tries to give each action a precise time and a precise list of the objects and resources it is going to use"[2]. In other words, plans resulting from plan generation become the inputs to scheduling. Scheduling is concerned with allocating resources, many of which - like time - are metric quantities. Control takes the output of scheduling, and either changes the world's state or triggers replanning or rescheduling to recover from execution failure. A recent development has been to consider plan generation and scheduling as concurrent tasks which place constraints on one another. In this thesis, I maintain the logical distinction between planning and scheduling to avoid the need to include metric quantities, dimensions and relationships in the POI ontology and algorithm.

My research centres on an "upstream" process (see Figure 5): the learning of sets of planning

[2] Similar distinctions are made by Georgeff (1987, pp. 381-382) and, in the Operations Research literature, by Salvador (1978, p. 268).

operators. There are parallel threads of upstream research into learning planning-system goals (e.g., see (Cohen and Levesque, 1987)) and control knowledge, marking the third era.

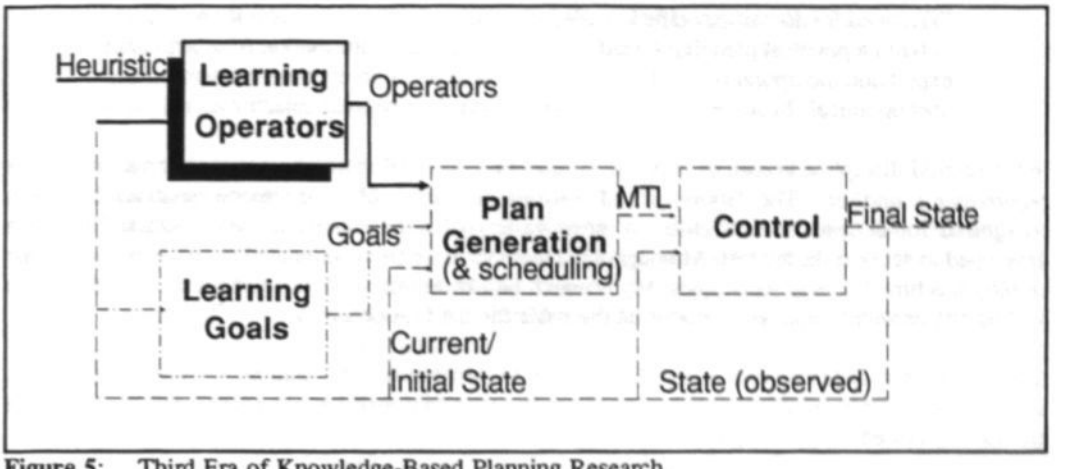

Figure 5: Third Era of Knowledge-Based Planning Research.

My research is independent both of the downstream processes and of the parallel upstream processes. Moreover, my research is not concerned with plan generation *per se*. It is influenced by plan generation only to the extent that the operators learned must be capable of input into plan generation. This implies that learning should induce an operator formalism that is widely used in plan generation. There is also an indirect feedback from Control, in that new states achieved by plan execution can be used as new inputs for further operator learning. Feedback is demonstrated in Chapter 5.

1.2 BACKGROUND

1.2.1 Related Fields

The question of where operators or similar data-structures come from has arisen previously in other fields, both inside and outside AI. In the expert systems subfield of AI, it has been known since the early 1980s that the compilation of sets of expert system rules is a difficult and laborious process. Compilation of knowledge is known as *knowledge acquisition*, and its difficulty has been termed the *Knowledge Acquisition Bottleneck* (Feigenbaum and McCorduck, 1984). A variety of methods have been developed to support the knowledge engineer in acquiring knowledge[3], some of which use machine-learning techniques[4]. Rule induction techniques are being actively studied, and have progressed to the stage at which rule-induction packages have been marketed. Although POI is not based on rule induction techniques, its purpose is similar, i.e., knowledge acquisition.

---

[3] The interested reader is referred to Hart (1989) and to McGraw and Harbison-Briggs (1989) for an introduction to knowledge acquisition. The wider area of knowledge engineering, with an emphasis on a structured approach to knowledge acquisition, is addressed by Greenwell (1988). Wielinga, Schreiber and Breuker (1992) describe KADS, arguably the current leading knowledge-acquisition methodology.

[4] Readings in knowledge acquisition, with an emphasis on applications of machine learning, may be found in Buchanan and Wilkins (1993).

Similar difficulties have recently been acknowledged to exist in the compilation of planning knowledge. For example, Minton and Zweben (1993, p. 2) state:

> "The need for domain-specific knowledge presents a serious problem if one intends to build a practical planning system. Typically, eliciting information from a domain expert and incorporating it into a planning system is extremely time-consuming and uneconomical. In some cases, no human expert may be available."

In the non-AI discipline known as software engineering, the analogous process is generally known as *requirements analysis*. The laborious and error-prone nature of requirements analysis has been recognised for at least three decades. A great variety of requirements-analysis methods has been developed to support the analyst. Although most methods have been automated as software tools, none employ machine learning techniques. My research builds on this wealth of experience by taking two leading requirements-analysis methods as the basis for the POI ontology.

In summary, the fields related to my research are knowledge-based planning, software engineering, and machine learning (with the emphasis on induction). Additionally, multi-agent systems techniques are used in closed-loop testing of the POI algorithm.

```
Part 1: Acquisition (optional)
Step (1.1): Acquire domain state description(s).
Step (1.2): Recognise objects and relationships from state descriptions.
Step (1.3): Compile constraints from objects and relationships.

Part 2: Induction
Step (2.1): Generate description language for domain.
Step (2.2): Construct version space using description language.
Step (2.3): Identify valid world states.
Step (2.4): Determine valid transitions using meta-heuristic.
Step (2.5): Generalise world states and transitions as (respectively)
            state-classes and transition-classes.
Step (2.6): Reformat transition-classes as planning operators.
```

Figure 6: Outline of POI Algorithm.

1.2.2 Outline of POI Algorithm

Figure 6 outlines the POI algorithm. At the heart of the algorithm (Steps (2.2) and (2.3)) is an extension of Mitchell's (1982) *version space and candidate elimination* algorithm for single-concept learning. In its POI application, the concepts to be learned are the valid states that can be exhibited by the domain. Domain states are learned by constructing a version space. In POI, the version space is a lattice of nodes, with each node being described in terms of the relationships between the domain objects. Validity of nodes and associated states is determined (i.e., "candidates" are "eliminated") by the interrelationship constraints. The valid states obtained by version-space construction must be post-processed (in Steps (2.4) to (2.6)) to extract transitions and to generalise the extracted transitions as planning operators. A meta-heuristic guides the extraction of transitions. Pre-processing may also be needed (in Steps (1.1) to (2.1)) to prepare the inputs for the learning process.

The nine steps of the POI algorithm are divided into two parts. Part 1 of the POI algorithm is named *Acquisition*, because it acquires a *domain model* consisting of unordered lists of objects, relationships and constraints from unordered observations of non-adjacent domain states. Acquisition is unnecessary

if input is already in this domain-model form. In the blocks world, the Acquisition process would take state descriptions such as[5]:

```
[[holding hand1 block1] [notOn block1] [notBeneath
block1] [notOnTable block1] [notSupporting table1],
```

and output the corresponding domain model:

Object-classes: [Hand, Block, Table].
Object-instances: [hand1, block1, table1].
Relationships: [holding, notHolding, notHeld, on, notOn, beneath, notBeneath, onTable, notOnTable, supporting, notSupporting].
Constraints[6]:
A hand cannot be holding and notHolding at the same time.
A block cannot be holding and on at the same time.
A block cannot be holding and beneath at the same time.
A block cannot be holding and onTable at the same time.
A block cannot be holding and notHeld at the same time.
A block cannot be onTable and notOn at the same time.
A block cannot be onTable and notBeneath at the same time.
A block cannot be onTable and notOnTable at the same time.
A block cannot be onTable and notHeld at the same time.
A block cannot be on and notOn at the same time.
A block cannot be on and notBeneath at the same time.
A block cannot be on and notOnTable at the same time.
A block cannot be on and notHeld at the same time.
A block cannot be on and beneath at the same time.
A block cannot be on and onTable at the same time.
A block cannot be beneath and notOn at the same time.
A block cannot be beneath and notBeneath at the same time.
A block cannot be beneath and notOnTable at the same time.
A block cannot be beneath and notHeld at the same time.
A block cannot be notHeld and notOn at the same time.
A block cannot be notHeld and notBeneath at the same time.
A block cannot be notHeld and notOnTable at the same time.
A table cannot be supporting and notSupporting at the same time.

Part 2 of the POI algorithm is named *Induction* because it embeds the computationally-expensive induction of the version space. The Induction process takes a domain model as input, either entered directly by the user or resulting from the Acquisition process. The domain-model relationships are instantiated with the object-instances, forming the description language used in building the version-space lattice. The domain-model constraints are used to prune the lattice during building. The result of lattice-building is the set of valid domain states. A meta-heuristic is used to identify the valid transitions between the induced domain states. Planning operators are extracted by generalising the transitions. For example, given the above domain model with several constraints invalidated by further observations, the Induction process should output the planning operators **pickup** and **putdown**.

---

[5] This description is expressed using the POI ontology to be introduced in Chapter 3.

[6] Several of these constraints would be invalidated by further observations, shortening the list.

The role of the meta-heuristic used in Step (2.4) can be best illustrated by an example. One possible meta-heuristic is Single-Actor/Single-State-Change (SA/SSC), which expresses the assumptions[7] that, during any given transition:

- there is just one object which initiates the transition: the single *actor*.
- the actor undergoes a change in just one of its relationships: the single *state-change*.
- there is a causal hierarchy of state-changes in the other participating objects, with the actor's state-change at the root of this hierarchy.

In effect, the SA/SSC meta-heuristic combines cause-and-effect with non-concurrent action. Other meta-heuristics permit an actor to undergo two or more changes (Multi-State-Change (MSC)) or two or more actors to initiate transitions simultaneously (Multi-Actor (MA)). All four meta-heuristics - SA/SSC, SA/MSC, MA/SSC, and MA/MSC - have been considered in my research.

1.2.3 Distinguishing POI from Plan Generation

Experience shows that planning operator induction can be confused with plan generation. An analogy helps to distinguish them. For convenience, the analogy will use the search paradigm for reasoning, and plan generation will be described first.

Feature in task	Feature in plan generation
Motorway network	Search-space
Junction	State
Drive on motorway	Planning operator
Stretch of motorway	Instantiated operator
Junction A	Initial state
Junction B	Goal state
Route	Plan
First agent	Planner, planning system, or planning algorithm
First agent's task	Plan generation

Figure 7: Correspondence between First Agent's Task and Plan Generation.

Suppose that an agent is given the task of planning a route through a network of motorways from junction A to junction B. A motorway network can be readily modelled as a graph with junctions as nodes and stretches of motorway as arcs. To find a route, the agent must search through the graph[8].

---

[7] Compare the STRIPS assumption, especially as described by Allen (1990, p. 51).

[8] The agent has the choice of representing either junctions or stretches of motorways as points in the search space. The latter choice corresponds to the usual representation of plans as sequences of actions. An example route would then be: "Go west along the A47, then south on the A11, then east along the M25, ...".

The agent has no map of the motorway network initially, but must construct one as search proceeds. This is (an analogy of) a typical plan generation task, with the correspondences listed in Figure 7.

By contrast, suppose that a second agent is given the task of mapping the Earth's surface, starting from the centre of the Earth[9]. The second agent has no geological information about the interior of the Earth. To locate the Earth's surface, the second agent must burrow (i.e., search) upwards until it breaks through into air. Each breakthrough point, together with the vertical route leading to it, is mapped. When all possible vertical directions from the centre of the Earth have been explored, the agent has a map of the Earth's surface in terms of the breakthrough points. The agent can then connect the breakthrough points together by applying a meta-heuristic. Planning operators are obtained by generalising these connections. This is an analogy of the Planning Operator Induction algorithm, with the correspondences listed in Figure 8.

Feature in task	Feature in Planning Operator Induction
Universe	Rule space
Earth's interior	Version space (lattice)
Earth's surface	State space
Breakthrough	Candidate elimination
Earth's centre	Bottom element of version space lattice
Breakthrough point	Point on *SG frontier*; state in state-space
Upward	Subsumption relation
Vertical route	Subsumption chain
Connection	Transition; instantiated operator
Generalised connection	Planning operator
Second agent	*POIAgent*
Second agent's task	Planning Operator Induction

Figure 8: Correspondence between Second Agent's Task and POI.

1.2.4 Potential Applications

The outline of the POI algorithm implies that both Parts are used. It is also possible to use the two Parts separately. The claim made in this thesis - that planning operators can be induced from objects, relationships and constraints - applies to the Induction part on its own. The use of Induction alone in order to induce planning operators from a domain model is illustrated extensively in Chapter 4. From this viewpoint, Acquisition can be regarded as a "front-end" to Induction.

Both parts of the POI algorithm are needed when the input is in the form of state descriptions. Although Acquisition extracts domain knowledge from the state descriptions, it is in a model-based form which not amenable to plan generation. Induction transforms the domain model into a form that

[9] This "mole's-eye" view of the Universe maps better onto version-space construction than does the more usual "god's-eye" view.

can be readily used for planning. The use of the full algorithm is illustrated in Chapter 5.

Acquisition could also be used on its own, e.g., to create an entity-relationship model and to list the domain constraints acquired from a set of observations of the domain. I have not investigated stand-alone application of Acquisition because it was unrelated to planning, and hence outside the scope of my research. However, the potential of decoupling Acquisition and Induction is also demonstrated in Chapter 5, where the domain model acquired by one agent is transmitted for induction by another.

POI could be applied as:

- *a knowledge acquisition tool for planners.* I foresee this as being the prime application, as illustrated by the experiments described in Chapter 4.

- *an element of multi-agent systems.* A POI-capable agent - a *POIAgent* - would be able to acquire knowledge about other agents in a form suited to planning. Using the acquired knowledge, the POIAgent would be able to solve problems which required the other agents to perform a sequence of actions. The experiments in Chapter 5 demonstrate POI in this role.

- *a Computer-Aided Software Engineering (CASE) tool.* Existing software engineering methods require software designers to develop entity-relationship models and state-transition networks separately. They are then cross-checked, either manually or with the help of a CASE tool. This cross-checking is a laborious process, which does not guarantee to eliminate all inconsistencies between the two representations. POI would enable the software designer to develop a state-transition network by inducing it automatically from the system's entity-relationship model, saving effort and guaranteeing consistency between the two representations. I have studied this application in the context of supporting the designers of spacecraft payloads (Grant, 1992c). This role is outside the AI field, and is not considered further in this thesis.

- *an intelligence support tool.* In competitive situations, commercial or military intelligence about other agents plays an important role. Ideally, information should be released by one agent so that the recipient agents acquire consistent, but false domain knowledge. POI could be used to model the effects of releasing items of information. The key feature is that POI would be used to reason about incorrect domain knowledge. I have not studied this application, because induction with incorrect information is outside the scope of my research.

- *an instruction support tool for teachers.* Information about other agents is also important in situations where agents are cooperating. An example is when a teacher instructs one or more students. POI could be used in seeing how a student's domain knowledge accumulates as he or she is presented with example problems. This would help in determining:

  - *whether the student had assimilated the examples completely.* POI could be used to predict what the student should know, given the examples presented.

  - *what example should best be presented next.* POI could used for "what-if" simulation of the knowledge gained by presenting each remaining example.

  The key feature is that POI would be used to reason about incomplete domain knowledge, which is also outside the scope of my research. However, the experiments in Chapter 5 can be interpreted as illustrating elements of the instruction support role.

1.2.5 Limitations

Of course, the POI algorithm has its limitations. In particular, the POI algorithm does not guarantee:

- *perfection*. It does not guarantee to induce a complete and correct set of planning operators from any set of world state descriptions or from any domain model. Given at least one state description, Acquisition will always generate a domain model. The domain model will only be as complete and correct as the state description(s) from which it was generated. Induction always induces one or more states from a domain model, but there may not be any valid transitions between them for the given meta-heuristic. When there are no valid transitions, then no planning operators can be extracted.

- *efficiency or optimality*. The version space and candidate elimination algorithm is known to be restricted to discrete attribute spaces, to noise-free data, and to two-class domains (Kalkanis and Conroy, 1991). It is "computationally expensive" (*ibid.*, p. 320), and the expense increases exponentially with the number of data (Haussler, 1988). Most seriously of all, it has a "serious problem ... when the target concept contains disjunctions" (Kalkanis and Conroy, 1991, p. 320), which lead to NP behaviour Murray (1987). As POI inherently involves the learning of multiple, disjunctive concepts, the POI algorithm is also NP. In the absence of better alternatives, I have devoted a major part of my research to developing *countermeasures*. These countermeasures do not eliminate the NP behaviour; they only postpone it. As the countermeasures enable the POI algorithm to be used in real-world domains (see Chapter 4), I regard their development as an integral part of my research.

- *applicability to all problems and all domains*. More efficient techniques (such as macro-operators (Fikes, Hart and Nilsson, 1972), case-based planning (Hammond, 1989), and sequence induction (Muggleton, 1990)) are available where pre-existing domain knowledge can be used. Where domain knowledge must be induced *ab initio* without recourse to sequencing information, then POI has no competitor. Even so, there are restrictions on the domains for which POI can be employed, e.g., where:

  - *inter-object relationships are non-metric*. This restriction arises from the version space and candidate elimination algorithm's limitation to discrete attribute spaces. One consequence is that POI is only applicable to planning, and not to scheduling or design. In practice, it may be possible to "discretize" metric relationships, but this is outside the scope of my research.

  - *inter-object relationships and interrelationship constraints are binary*. In practice, this is not a serious restriction, because any discrete domain can be modelled using binary relationships and constraints by careful choice of the domain objects and relationships (Frost, 1986). In Chapter 4 an example is given where this restriction is overcome by modifying the domain model. In short, it is a modelling problem.

1.3 APPROACH

1.3.1 General Approach

My research has been driven by the desire "to extend the range of things computers can do" (Bundy, Burstall, Weir and Young, 1980, p. ix). In other words, I take the *engineering approach* to AI. Although I build a computational model of a particular cognitive process - the learning of planning operators - I make no claims for its cognitive validity[10].

I go beyond the meaning of the phrase "engineering approach", as normally used in the AI community, in that I am concerned in this thesis with engineering design. To this end, I make more use of software engineering ideas and methods than is usual in AI research. In particular, I have used the industrial-strength entity-relationship model and state-transition network representations, supported by a computer-aided software engineering (CASE) tool[11]. Justification is given in Chapter 2.

1.3.2 Specific Exclusions

There are many aspects of knowledge-based planning, software engineering, and machine learning which are specifically excluded from my research. I am not concerned in this thesis with:

- scheduling, design or other specialisations of planning concerned with metric quantities;
- uncertainty (and associated handling methods) arising from forgetful agents, from imprecise, incorrect, or conflicting domain knowledge, or from incompleteness in initial or goal state descriptions.
- the selection of observation-series, e.g., for the purposes of supervised learning;
- the recognition of objects and their parent classes in world-state observations. For the purposes of this thesis, I assume that an object and its class can be recognised by using suitable naming conventions (see next section).

1.3.3 Choice of STRIPS

I employ the STRIPS formalism (Fikes and Nilsson, 1971) as an illustrative representation for planning operators both because it is simple to implement and because it and its derivatives predominate in knowledge-based planning (Georgeff, 1987). Other formalisms are more expressive, and could be accommodated by suitably modifying the final step in the POI algorithm.

---

[10] The cognitive validity of the planning operator representation itself has been questioned; see (Suchman, 1987).

[11] EasyCASE version 4.0, from Evergreen CASE Tools, Inc.

1.3.4 Research History

This section summarises the history of my research in chronological order. Inevitably, my studies have included investigations along avenues which have later turned out to be irrelevant. For clarity, only the relevant history is described. The thesis chapter order shows the benefit of hindsight.

My research programme began by identifying planning and scheduling as suitable research areas. I investigated the setting in which a typical real-world planning task was performed by human experts. My investigations showed that planning was almost always performed using uncertain information. The predominant form of uncertainty was incompleteness.

The second stage in my research programme was to investigate systematically the behaviour of a typical knowledge-based planning system when its input information is correct, but incomplete. I investigated incomplete sets of planning operators separately from incomplete planning problems. As I expected, I found that, for some problems with incomplete information, the planning system was incapable of generating any plan, and for others, the planning system was capable of generating complete and correct plans. But I also found that there were problems for which the planning system generated *invalid plans*, i.e., plans that cannot succeed because, on execution, they would either violate one or more domain constraints, or stop short of achieving the desired goal state, or use resources inefficiently (e.g., by performing some action and then undoing it again). Inspection of the invalid plans showed that some of them would have been valid in variants of the illustrative domain used. The variant domains differed in that, in effect, one or more domain constraints had been relaxed.

In third stage, I developed a small program which took lists of domain objects and constraints as its inputs and which delivered state descriptions as its output. Known as "BOOTSTRAP", the program was intended originally as a utility for systematising the generation of the incomplete state descriptions to be used in plan generation. BOOTSTRAP used a non-iterative version of Mitchell's (1982) version spaces and candidate elimination algorithm.

The fourth stage occurred when I realised that BOOTSTRAP could be extended to output planning operators, rather than state descriptions[12]. At the same time, I noticed that the domain constraints, as represented in BOOTSTRAP, could be used to select a suitable action in response to a particular world state, i.e., for *reactive planning*.

In the fifth stage, I incorporated the extended BOOTSTRAP program in a multi-agent environment. The resulting system, known as the Message-Based Architecture (MBA) testbed, integrates reactive planning, plan generation, and induction of planning operators (Grant, 1991). MBA agents perform *learning-by-doing* (Anzai and Simon, 1979). The MBA testbed was extended so that agents could exchange acquired knowledge, i.e., they could also *learn-by-being-told*. The MBA system and experiments are described in Chapter 5.

The sixth stage in my research was to extract the integrated reactive planning, plan generation and induction functionality implemented in the MBA agents into a single-agent system, known as the Dutch Utilisation Centre's Activity Scheduling System (DUC-ASS). The induction algorithm was refined, using Reason Maintenance System techniques, to reduce its requirements for computer memory and run-time. Countermeasures were developed to combat the combinatorial explosion. The system functionality was enhanced so that it was possible to induce procedures (i.e., skeletal or generalised plans), as well as planning operators. Useability was improved by implementing a

---

[12] This was the *eureka* moment in my research.

graphical user interface and a facility to generate multi-media documentation compatible with European Space Agency standards[13]. DUC-ASS has been used to induce planning operators for many domains, including the High Performance Capillary Electrophoresis spacecraft payload (Eckhard, 1992) intended to be flown in the International Space Station. The induction algorithm, as in DUC-ASS, is documented in Chapter 3. The DUC-ASS program and experiments are described in Chapter 4. With the advantage of hindsight, the single-agent experiments using DUC-ASS have been documented in this thesis before the MBA multi-agent experiments.

1.4 THESIS CONVENTIONS AND LAYOUT

1.4.1 Naming Conventions

I use the following naming conventions:

- Entities (i.e., objects, relationships, and operators) have single-word names, with phrases being compounded into a single word, e.g., "Pressure Supply" would become "PressureSupply". Domain object- and relationship-names are in `Courier` font, e.g., `Block`, `hand2`, `Finger`, `PressureSupply`, `vialReceptacle3`. Operator-names are also in bold, e.g., **`pickUp`**, **`putDown`**, **`stack`**, **`unstack`**.

- The names of classes (i.e., of objects, relationships, states, and transitions) and global variables (in Smalltalk code) have an initial capital and contain no numeric digits, e.g., `Block`, `Finger`, `PressureSupply`, etc.

- The names of instances (i.e., of objects, relationships, states, and transitions) are formed from the name of their parent class by converting the initial letter to lower-case and suffixing the resulting word with a numeric digit unique to that class. For example, the `Block` object-class could have object-instances named `block1`, `block2`, `block3`, and so on. Conversely, an agent encountering an object named `uVDetector4` can deduce that the object is an instance of the `UVDetector` class, even if that class was previously unknown to the agent. This convention enables me to exclude the *object recognition* problem from the scope of my research.

1.4.2 Thesis Layout

There are six chapters in this thesis. Chapter 2 reviews the relevant literature from the fields of software engineering, knowledge-based planning, machine learning, and multi-agent systems. Chapter 3 documents the POI ontology, algorithm, and implementations. Chapters 4 and 5 describe experiments with the two implementations, in open- and closed-loop testing, respectively. Chapter 6 summarises the thesis, draws conclusions, lists some issues for possible study by other researchers, and identifies the novel contributions of my research. Additional material includes a bibliography, an index, summaries in English and Dutch, and my curriculum vitae.

---

[13] I am grateful for the partial funding from the Nederlands Instituut voor Vliegtuigontwikkeling en Ruimtevaart (The Netherlands Agency for Aerospace Programs) under contract NRT 2103B ("Pilot Dutch Utilisation Centre") for this part of the sixth stage.

2 LITERATURE REVIEW

The purpose of this chapter is to review the relevant literature in the fields of software engineering, knowledge-based planning, machine learning, and multi-agent systems. Software engineering and knowledge-based planning are the sources of the knowledge representations used in POI. Machine learning is the source of POI's core inferencing process. Multi-agent systems techniques are used in closed-loop testing of the POI algorithm. Within a given field, the emphasis is on those topics that are directly relevant to my research. No attempt is made either to summarise the historical development or to provide complete coverage of all aspects of a field. References are provided to more comprehensive reviews.

First, I review the software-engineering field, introducing the entity-relationship model and state-transition networks on which the POI ontology (defined in Chapter 3) is based. Second, I review the AI field of knowledge-based planning, relating the knowledge representations used in plan generation to the entity-relationship model and state-transition networks. Third, I review inductive inferencing in machine learning, focusing on the version space and candidate elimination algorithm and on other researchers' approaches to operator learning. Fourth, I review the multi-agent systems literature on learning, emphasising the learning of planning knowledge in a multi-agent context. Finally, the chapter is summarised and its contributions highlighted.

2.1 SOFTWARE ENGINEERING

Software engineering is defined as the application of a systematic, disciplined, and quantifiable approach to the development, operation, and maintenance of software (Marciniak, 1994). Blum (1992, p. 10 and p. 20) states that software engineering involves "the application of tools, methods, and disciplines to produce and maintain an automated solution to a real-world problem". This review concerns only software engineering methods, because tool use is sensitive to commercial influences, and disciplines relate to the management of the software development process.

There are three sub-sections. First, the types of software engineering method are summarised. Then two methods - entity-relationship modelling and state-transition networks - are introduced.

2.1.1 Software Engineering Methods

Software engineering methods are defined as representations of software designs together with the procedures for performing inference with those representations. Graphical notations are often employed in documenting the representations. There is an extensive literature on methods; see Blum (1992) and Freeman and Wasserman (1984).

There are two types of software engineering method:

- *Formal methods* consist of precisely-defined notations and proof rules derived from set theory. A typical formal method provides for the definition of data objects and operations, refinement of the definitions by top-down decomposition, and proof rules to ensure that high-level and decomposed representations correspond. Formal methods excel in specifying functional behaviour and in deriving a correct design from specifications. However, the practical use of formal methods tends to be limited because formal specifications are hard

to write for large systems (Gehani, 1982).

- *Empirical methods* consist of notations and inference procedures which have been derived from practical experience in software development. They have been "proven" by large numbers of software designers, rather than by formal means. Some empirical methods - including, notably, entity-relationship models and state-transition networks - do have a rigorously-defined basis. All mature empirical methods provide one or more notations, an explicit decomposition approach, modularity, and an environmental model which defines the system's boundaries and interfaces. Consistency rules are used for the selection and naming of objects and their relationships. Informal procedures are given for checking the self- and mutual-consistency of the data and transformation schema. Most empirical methods give guidance on requirements capture and can express non-functional requirements such as user interfaces and project management.

I adopt empirical methods in this thesis, because the POI algorithm and several of the domains are large systems.

Empirical methods can be divided into structural and behavioural categories. *Structural* methods concentrate on describing the static structure of a software system, i.e., its decomposition into modules, routines, programs, packages, and the like. *Behavioural* methods concentrate on describing the dynamic behaviour of the software system, e.g., what outputs result for a given input, module-calling sequences, and state-transition networks. For completeness, both the static and dynamic characteristics of a software system must be specified. The current practice in Software Engineering is to model the software system twice: once using a structural method, and once using a behavioural method. The structural and behavioural models are compared, and then modified as necessary to ensure that they are consistent with one another. By contrast, POI generates the behavioural model of a domain directly from its structural model.

There are two fundamental orientations in structural methods:

- *Process-oriented methods* focus on the algorithmic abstractions of the problem domain. These methods lead to program modules that are highly functional, and are best suited to implementation in a procedural programming language.

- *Data-oriented methods* focus on the data abstractions of the problem domain. These methods lead to program modules that are highly declarative.

Since declarative programming is better suited to AI applications, I adopt a data-oriented method - the *entity-relationship model* - for describing the structure both of the POI algorithms and of the domains.

I adopt *state-transition networks* for describing behaviour.

2.1.2 Entity-Relationship Modelling

There are four main data-oriented models (Chen, 1976): the network model, the entity-set model, the relational model, and the entity-relationship model. In his seminal paper, Chen presented the entity-relationship model and showed how it could be used to unify the other three data models while maintaining their advantages. We are concerned here with the entity-relationship model *per se*, and not with its comparison with other data-oriented models.

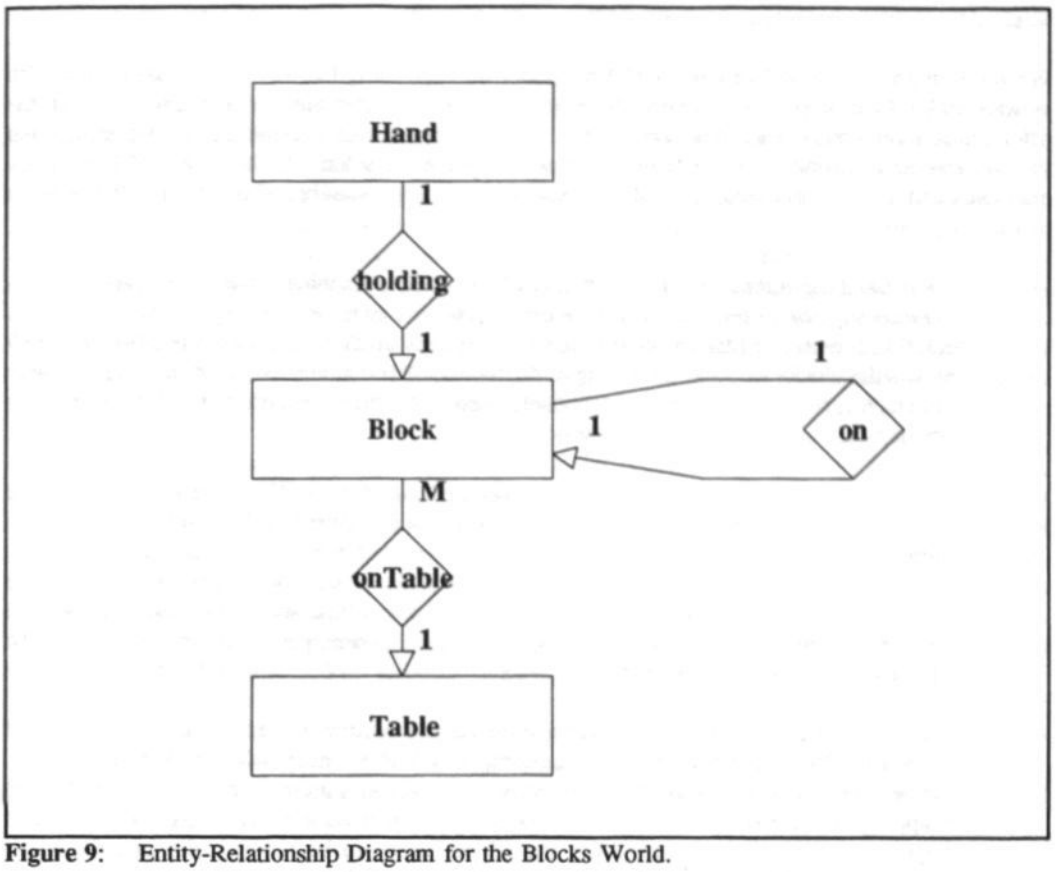

Figure 9: Entity-Relationship Diagram for the Blocks World.

The entity-relationship model is rigorous, being based on relational theory, which is itself based on set theory. Chen's (1976) version of the entity-relationship model distinguishes entities, entity-sets, relationships, relationship-sets, attributes, values, value-sets, and roles. He defines an *entity* as a "thing"[14] which can be distinctly identified. Entities are classified into *entity-sets*; there is a predicate associated with each entity-set to test whether an entity belongs to it. A *relationship* is an association among entities. A *relationship-set* is a mathematical relation among *n* entities, each taken from an entity-set. Each (ordered) tuple of entities in a relationship-set is a relationship[15]. The *role* of an entity in a relationship is the function that it performs in the relationship. *Values* are data items, such as "3", "red", "Peter" and "Johnson", and are classified into different *value-sets*, such as FEET, COLOUR, FIRST-NAME, and LAST-NAME[16]. There is a predicate associated with each value-set to test whether a value belongs to it. Information about an entity or relationship is obtained by measurement or observation, and is expressed by a set of attribute-value pairs. An *attribute* is defined as a function which maps from an entity-set or a relationship-set into a value-set or a Cartesian product of value-

[14] His quotation marks.

[15] Although Chen (1976) did not explicitly say so, I take it as read that there is a predicate associated with each relationship-set to test whether a relationship belongs to it.

[16] These are Chen's (1976) own example values and value-sets.

sets.

We are primarily interested in Chen's (1976) model for organising the information associated with entities and relationships, i.e., Chen's "level (1)". His key contribution was to propose that the information about entities should be separated from the information about relationships. He argued that this separation is useful in identifying functional dependencies among data. The POI ontology maintains and extends this separation of information. There are a number of other important aspects of Chen's paper:

- *He defined a graphical notation.* Chen (1976) defined a graphical notation, known as *entity-relationship diagrams*, in which entity-classes (drawn as rectangles) are linked to relationship-classes (drawn as diamonds with attached arcs). Using Chen's notation, Figure 9 shows the blocks world as consisting of the `Hand`, `Block`, and `Table` entity-classes, with `holding`, `on`, and `onTable` relationship-classes. In this thesis, entity-relationship models are depicted using Chen's graphical notation.

- *He introduced cardinality constraints on relationship-classes.* There are many alternative notations for cardinality (Blum, 1992, p. 108). I adopt Chen's (1976) notation in which cardinality constraints are shown as numbers at the extremities of the relationship-class arcs. The cardinality constraints shown in Figure 9 indicate that `holding` is a one-to-one relationship, i.e., that a hand can only hold one block at a time and a block can only be held by one hand at a time. Similarly, `onTable` is many-to-one, permitting many blocks to be on a table at a time, but restricting each block to being on only one table at a time.

- *He showed that attributes and relationships can be used to identify entities.* Level (2) of Chen's (1976) model concerns the representations of the conceptual objects from level (1). He represents entities by means of an *entity key*, which is a group of attributes such that the mapping from the entity-set to the corresponding group of value-sets is one-to-one. If several keys exist, the entity's primary key should be chosen to be semantically meaningful. If necessary, "we may define an artificial attribute and a value set so that such a mapping is possible" (*ibid.*, p. 14). In certain cases, the entities in an entity-set cannot be uniquely identified by the values of their attributes. One or more relationships must be used to identify such entities; Chen terms entities identified in this way as *weak entities*. Strictly speaking, therefore, Chen proposed an *entity-relationship-attribute* data-oriented model, but showed that it is possible to replace attributes by relationships (or vice versa). The POI ontology takes the replacement of attributes to one extreme[17], so that it is a *pure entity-relationship model*[18].

In the context of POI, Chen's (1976) model exhibits several restrictions. Some can be overcome by extending Chen's model. The restrictions are:

- *Terminology.* Chen (1976) is unclear about what an entity or entity-set is. I use the term "object", in the sense used in object-oriented analysis and design, instead of Chen's term "entity". Moreover, I use the suffix "class" instead of Chen's suffix "set", with the suffix "instance" being used to distinguish class-members from classes. The predicates associated with testing membership of the entity-sets and relationship-sets are replaced by class-

[17] The entity-attribute-value model - employed in most object-oriented languages - is the other extreme.

[18] The name of an object or a relationship being the only attribute which is not replaced by a relationship.

membership tests. Hence, in the POI ontology there are "object-classes", "object-instances", "relationship-classes", and "relationship-instances"[19].

- *No modelling of generalisation.* Generalisation is the modelling process in which differences between similar objects are ignored to produce a higher-order object (Smith and Smith, 1977). In the POI ontology, generalisation is the production of a higher-order object-class (the *superclass*) by ignoring differences between similar, lower-level object-classes (the *subclasses*). The superclass is linked to each of its subclasses by a `generalisationOf` relationship; superclass and subclass are the roles of this relationship. Since a given object-class can simultaneously play the roles both of superclass and of subclass, the `generalisationOf` relationship defines a *generalisation-specialisation* hierarchy. Although Chen's (1976) version of the entity-relationship model did not model generalisation, there are versions which have been extended to incorporate generalisation, e.g., Nijssen's Information Analysis Methodology (Nijssen and Halpin, 1989).

- *Modelling of aggregation.* Chen's (1976) model permitted second-order relationships, i.e., relationships between relationships. An alternative representation, known as *aggregation* (Smith and Smith, 1977), is to regard a (first-order) relationship as a higher-level entity. This entity can then be related to other entities. In the POI ontology presented in this thesis, second-order relationships are not permitted, and aggregation is not supported. Aggregation can be modelled using the existing POI ontology by identifying an additional domain object to represent the aggregate[20], with `partOf` relationships linking it to the objects modelling its constituent parts. The aggregate object fills the *parent* role, and the objects which are related to it by `partOf` fill the *child* role. Since a given object can be both an aggregate and a part of another object, the `partOf` relationship defines a *whole-part* (or *decomposition*) hierarchy. Ideally, the POI ontology should provide an appropriate representation for decomposition, as for generalisation-specialisation. However, this was not essential for my research.

- *Modelling other hierarchical relationships.* In addition to generalisation and aggregation, there may be other relationships that define hierarchies. Hierarchical relationships have the characteristic that they relate an object-class to itself (or to a superclass, subclass, parent, of child object-class). For example, the relationship `Person isParentOf Person` defines the type of hierarchy usually known as a *family tree*, and the relationship `Person isBossOf Person` defines an *organisational hierarchy*. In the blocks world, the relationship `Block on Block` defines hierarchies - stacks - of blocks. Since hierarchical relationships other than generalisation and aggregation are domain-specific, they can be modelled using the existing POI ontology.

- *Inability to express all interrelationship constraints.* Chen's (1976) Entity-Relationship Diagrams cannot express all interrelationship constraints. For example, Figure 9 does not show that a block which is participating in a `holding` relationship cannot at the same time participate in an `onTable` relationship. Such a constraint is known in Nijssen's Information Analysis Methodology (Nijssen and Halpin, 1989) as an *exclusion constraint*. The POI ontology extends Chen's entity-relationship model to express both cardinality and exclusion

---

[19] And "state-classes", "state-instances", "transition-classes", and "transition-instances".

[20] The introduction of additional domain objects (and, if necessary, object-classes) can be justified by appeal to Chen's (1976) analogous introduction of additional attributes and value-sets.

constraints between relationships. Unlike Chen's model, the POI ontology restricts constraints to being binary. Frost (1986, section 3.4.12, pps 111-118) shows that there is no loss in generality, providing that additional domain objects can be identified as necessary[21].

- *No modelling of mandatory and optional roles.* A role is *mandatory* if it must exist for every object-instance in all world states (Nijssen and Halpin, 1989). Chen's (1976) entity-relationship model makes no explicit distinction between mandatory and optional relationships. By contrast, all relationships in the POI ontology are optional by default[22]. For example, it is not mandatory in the blocks world that a `Hand` shall be `holding` a `Block`; it could be empty. The disadvantage of optional relationships is that it is not possible to make the Closed World Assumption (Reiter, 1978). To counteract this disadvantage, the POI ontology provides converse and inverse relationships. A *converse* relationship is simply a renaming of the primary relationship in the reverse direction, e.g., `[heldBy Block Hand]` would be the converse of `[holding Hand Block]`. An *inverse* relationship is the logical opposite of its primary. There is one inverse relationship for each role associated with the *primary* relationship, e.g., `[notHolding Hand]` would indicate the absence (for the Hand) of `[holding Hand Block]`. By providing an inverse relationship, it becomes possible to flag as incomplete any blocks world in which there is a `Hand` which is neither `holding` nor `notHolding`. There would be a similar `notHeld Block`-inverse relationship.

In summary, the POI ontology is obtained by extending Chen's (1976) entity-relationship model in the following ways:

- *Object-relationship model.* The basis for the POI ontology is an object-relationship model, with a clear distinction between classes and instances. By default, all relationships in the POI ontology are optional.

- *Inheritance.* The POI ontology includes a generalisation-specialisation (inheritance) hierarchy of object-classes. The POI ontology could also include a whole-part hierarchy of object-classes, but this was not found to be essential for research purposes.

- *Interrelationship constraints.* The POI ontology expresses exclusion constraints between relationships, as well as Chen-style cardinality constraints. All constraints in POI are binary.

2.1.3 State-Transition Networks

Software systems have the key property that, when executed, they exhibit dynamic behaviour. For this reason, all empirical software design methods include a notation to describe the pattern of changes in the software system over time. The leading notation is the *state-transition network.*

A state-transition network is a rigorous notation that is derived from automata theory (von Neumann, 1966), itself derived from the Turing machine (Turing, 1936). A system can be abstracted as an automaton that moves from one state to another, where a *state* is an aggregate of the system's attribute-values, including inter-object relationships. The dynamic behaviour of the system can then

---

[21] Again justified by appeal to Chen's (1976) analogous introduction of additional attributes and value-sets.

[22] Mandatory roles can be represented by suitable choice of exclusion constraints.

be modelled as a series of events that cause the model to proceed from one state to another. State-changes may be modelled as occurring continuously or discretely. In this thesis, state-change will be assumed to be discrete and instantaneous, i.e., state-change occurs by means of *transitions*. A transition is triggered by an input event, and generates a set of output events.

In the POI ontology, the automata are the object-instances in the domain, each having sets of states, transitions, and triggering events characteristic of its object-class. Object-instances interact by passing messages, so that, in POI, an event is modelled as the transmission of a message from one object to another.

There are two leading models of automata: the Mealy (1955) model, and the Moore (1956) model. Both represent an automaton as a 5-tuple, consisting of a set of inputs, a set of outputs, a set of states, a "next-state" function, and an output function. In the Mealy model, outputs are a function of inputs and states. An output signal persists only so long as its corresponding input and state are present. By virtue of the "next-state" function, the automata changes its state immediately on receipt of the input. Hence, the output signal is an instantaneous event in the Mealy model. By contrast, in the Moore model, outputs are a function only of states. Since states may be persistent, Moore-model outputs may also be persistent. The POI ontology adopts the Mealy model, because transitions and their triggering events are required to be instantaneous, with both inputs and outputs as events.

In general, the states and transitions of a domain form a network: the *state-transition network*. A state-transition network is depicted graphically as a state-transition diagram (Shlaer and Mellor, 1992). By software engineering convention, states (shown as boxed nodes) are linked by transitions (shown as squared arcs). Each transition is labelled with the event which triggers it, with events being denoted by a short line orthogonal to the relevant arc. Figure 10 shows the complete state-transition network for a one-hand, one-block, one-table world.

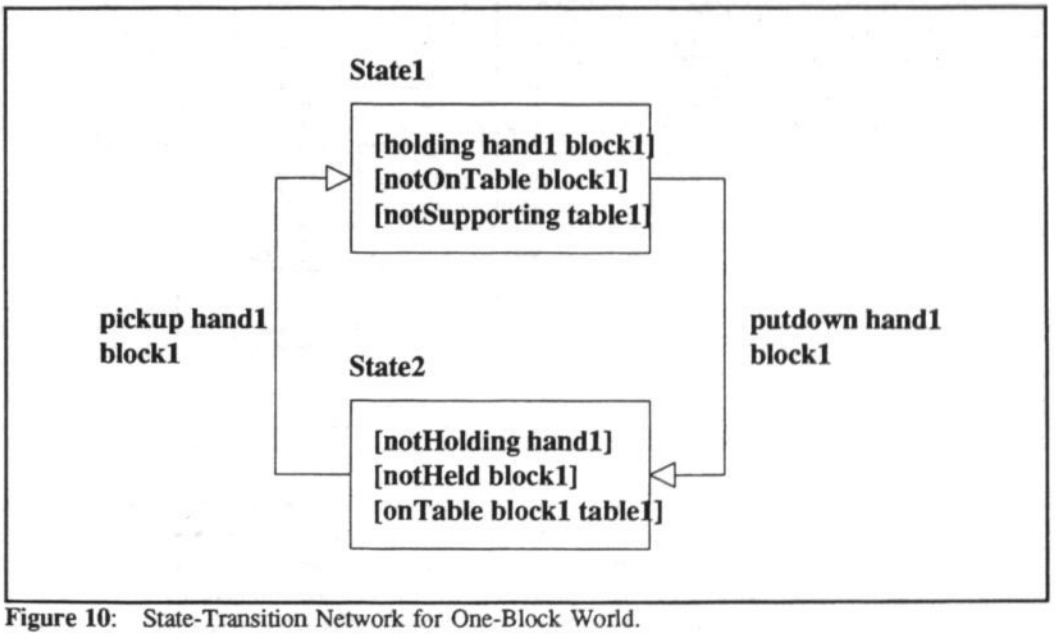

Figure 10: State-Transition Network for One-Block World.

There are practical difficulties in drawing state-transition networks for complex systems. Typically, there are too many states and transitions to depict in a single diagram. Three common solutions have

been adopted in software engineering; all three are applied in POI. The solutions are to:

- *Generalise states and transitions.* As in generalising objects and relationships, states and transitions can be generalised by ignoring differences between instances. For example, Nilsson's (1980) diagram of the state-space for the three-blocks world (*ibid.*, p. 283, Figure 7.3) can be generalised as shown in Figure 11. Nilsson's diagram is shown on the left-hand side (Figure 11(a)). In the POI ontology, generalised states are known as *state-classes*, and generalised transitions are known as *transition-classes*. The set of state-classes corresponding to Nilsson's diagram, shown on the right-hand side (Figure 11(b)), has been obtained by gathering together states with the same configurations and consistently replacing names of object-instances by names of object-variables. In (b), the object-instances A, B, and C in the right-hand-most path down Nilsson's diagram have been replaced by object-variables b_1, b_2, and b_3, respectively. The POI ontology provides for state-classes and state-instances and for transition-classes and transition-instances.

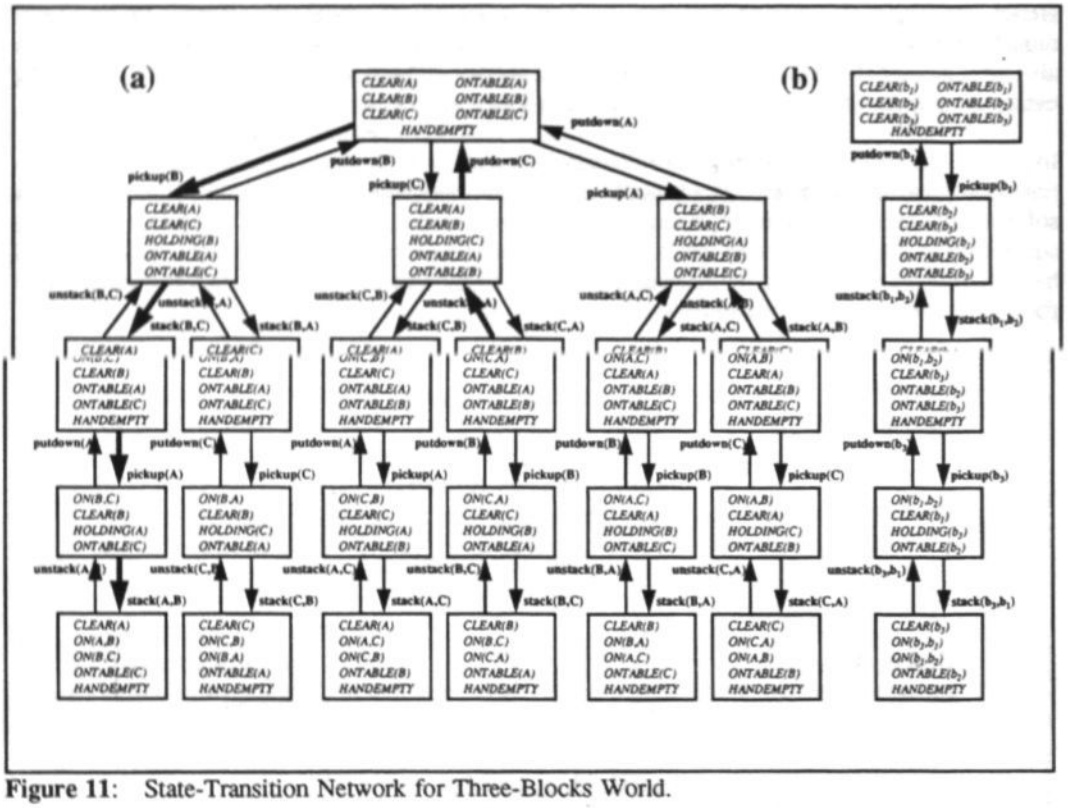

Figure 11: State-Transition Network for Three-Blocks World.

- *Partition states and transitions.* A state-transition network (STN) can be partitioned in several different ways. A state-transition network partition will be termed a *sub-STN*. Partitioning can be done by object-classes (to give a set of *class-STNs*) or by object-instances (to give a set of *instance-STNs*). The unpartitioned state-transition network will be termed the *global-STN*. For example, Nilsson's (1980) three-blocks world state-transition network (Figure 11(a)) depicts a global STN that can be simplified by partitioning by object-class. Figure 12 shows the resulting class-STN for the `Hand` object-class. Partitioning by object-class is often found in commercially-available CASE tools that support the drawing of state-

transition networks. In the POI algorithm, partitioning gains extra significance, because it reduces the size of the version-space lattice underlying the states in the state-transition network. For this reason, partitioning is a key countermeasure to the combinatorial explosion.

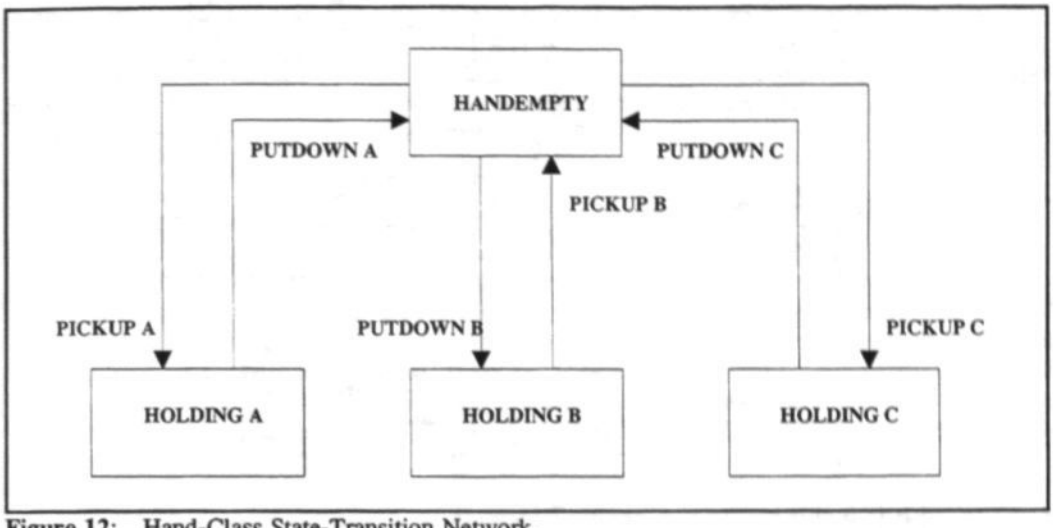

Figure 12: Hand-Class State-Transition Network.

- *Compose states and transitions.* Composition is closely coupled to the granularity of domain representation chosen by the planning system's designer. A frequently-occurring sequence of states can be composed into a single, larger-grain state. For example, Nilsson's (1980) three-blocks-world state-transition network (Figure 11) can be simplified by composition; each state in which the hand is empty is composed with the adjacent state in which the hand is holding the block removed from the top of a stack. The result is to reduce 22 states to 13 (see Figure 13), and is identical to the state-space depicted in Genesereth and Nilsson (1987, Figure 11.4, p. 269)[23]. In this case, composition has been achieved by omitting the `Hand` object-class from Genesereth and Nilsson's representation of the blocks world. Unlike generalisation and partitioning, composition cannot be exploited within the POI ontology and algorithm. However, the POI algorithm can be used by the planning system's designer to investigate the effects of varying the domain representation's granularity. An over-large state-transition network would indicate that the domain representation was too fine-grained.

[23] Genesereth and Nilsson (1987) make no mention of any relationship between their Figure 11.4 and Nilsson's (1980, p. 283) Figure 7.3, despite there being an author in common.

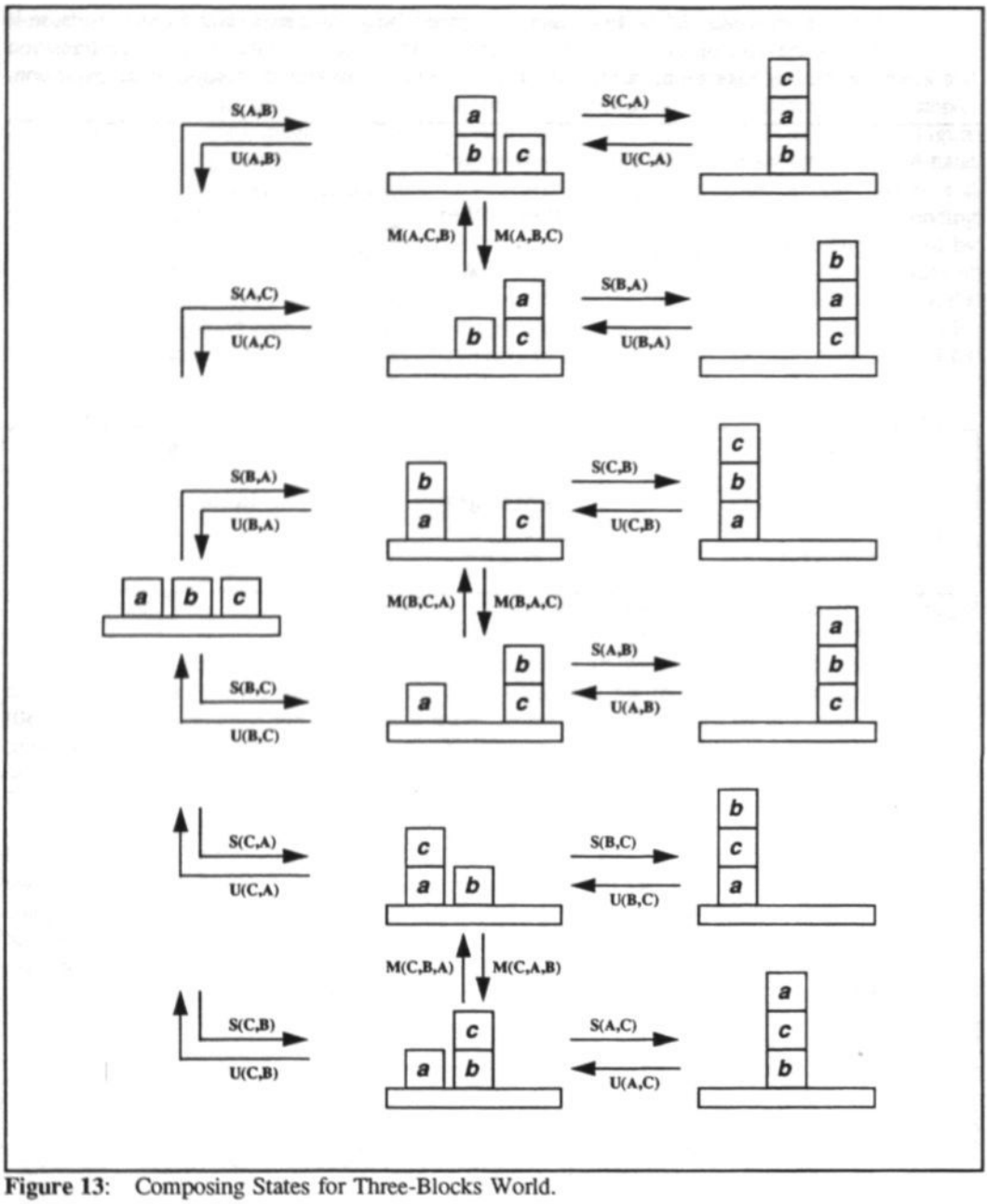

Figure 13: Composing States for Three-Blocks World.

2.2 KNOWLEDGE-BASED PLANNING

This section builds on the previous section's introduction of the entity-relationship model and the state-transition network. The aim is to relate these software engineering methods to knowledge representations found in plan generation so as to provide the foundation for the POI ontology.

There are seven sub-sections. First, the development of ontologies in AI is sketched. Second, the types of knowledge found in plan generation are identified. Then the representations of five groups of knowledge-types are reviewed in more detail.

2.2.1 Development of Ontologies

In AI, a *knowledge representation* is "a combination of data structures and interpretative procedures that, if used in the right way in a program, will lead to 'knowledgeable' behaviour" (Barr and Feigenbaum, 1981, p. 143, quotes in original). Ever since the 1970s, AI researchers have recognised that acquiring knowledge about a domain is the key to building large and powerful knowledge-based systems. There are two steps in the knowledge acquisition process: eliciting knowledge from a domain expert or other source, and representing it in a suitable executable formalism. Barr and Feigenbaum's (ibid., p. 8) contributors note that:

> "[A] basic open question involves the original conceptualization of a problem, called in AI the choice of problem representation. Humans often solve a problem by finding a way of thinking about it that makes the solution easy - AI programs, so far, must be told how to think about the problems they solve".

Hence, an important aspect of AI research is the discovery of better knowledge representations for given tasks.

A means for documenting knowledge representations is an *ontology*, which is a set of definitions of content-specific knowledge representation primitives (e.g., classes, relations, functions, and object constants) (Gruber, 1992). Ontologies are primarily intended for sharing domain knowledge between knowledge-based systems (Fikes, Cutkosky, Gruber and Van Baalen, 1991). For example, a common ontology of the blocks world could be developed for use by a vision system, a planning system, and a robot. The vision system would use the ontology in describing the current configuration of a real set of blocks, the planning system would use the ontology in generating a plan for achieving a goal configuration, and the robot would use the ontology in executing the plan.

Domain ontologies are not useful directly in POI. Definition of the knowledge representation primitives for the blocks world is of little use when inducing planning operators for the Dining Philosophers Problem. A meta-level ontology, i.e., one that is generic to multiple domains, is needed for POI. For example, it is useful to know that many domains, including the blocks world and the Dining Philosophers Problem, consist of objects which have states that can be changed by executing operators. Gruber (1992) terms such meta-representations *common ontologies*.

A key aspect of my research has been to develop a common ontology suited to the induction of planning operators: the *POI ontology*. The POI ontology is a domain-independent meta-representation for modelling the domain-specific knowledge used in inducing an operator-set. It must marry a knowledge representation used for modelling domains (the "structural model") to a knowledge representation used for plan generation (the "behavioural model"). The link is made by exploiting a knowledge representation found in machine learning (the "linking model"). More details are given in

the next chapter.

A common ontology for planning and scheduling is currently under development as a part of the US ARPA/Rome Laboratory Planning Initiative. The first iteration, named the *Knowledge Representation Specification Language* (KRSL), was completed in early 1994. The current iteration is known as the *Planning Ontology Project* (POP). Few details of the ARPA/Rome Laboratory Planning Initiative have been published in conferences, although some material is publically available through Internet[24]. Inspection of the publically-available plan ontologies shows that they overlap with POI's behavioural model, but do not cover the equivalent of POI's structural or linking models.

The POI ontology is not alone in being a meta-representation. Ontolingua (Gruber, 1992) is a language for writing ontologies in a canonical format such that they can be easily translated into a variety of representation and reasoning systems. The organisation of knowledge in Ontolingua is based on classes and instances of objects, on sets, on relations, and on assertions whose meaning depends on the contents of the knowledge base. The Relationship Lattice Laboratory (RLL) (Pedersen, 1994) is a tool for the construction of multimedia software systems for the retrieval and presentation of information from heterogeneous sources. The organisation of knowledge in RLL is based on relationship lattice theory, in which concepts (each with an extension and an intension) are linked in a lattice of relationships. The extension of a concept (cf. object-class) represents the set of individuals (cf. object-instances) in the modelled universe. The intention is a set of properties, given by the relationships. Relationship lattice theory can model itself, i.e., it is a meta-representation.

It is not enough to provide a common ontology for POI. An ontology merely describes a static set of terms. The dynamic, problem-solving procedures using those terms must also be defined. In POI, the dynamic, problem-solving procedure is known as the *POI algorithm*, and is also detailed in the next chapter. In software engineering terminology, the POI ontology is comparable to a structural method, and the POI algorithm is comparable to a behavioural method.

2.2.2 Knowledge Representations in Plan Generation

To develop an ontology for inducing planning operators, it is first necessary to identify the knowledge representation primitives used in plan generation, together with the associated terms and representations. This can be done by inspecting a suitable definition of plan generation from the literature. Where necessary, the representations will be related to equivalent representations used in software engineering.

The AI literature on plan generation is extensive. Introductory material on plan generation may be found in a number of AI textbooks, including Nilsson (1980, chapters 7 and 8), Bundy, Burstall, Weir and Young (1980, pp. 42-58), Rich and Knight (1991, chapter 13), Charniak and McDermott (1985, chapter 9), and Winston (1979, pp. 287-301; 1984, pp. 228-236; 1993, pp. 323-346). There are also some books on plan generation, but these are generally either PhD theses (e.g., Sacerdoti (1977), Suchman (1987), and von Martial (1992)), or descriptions of particular planning systems (e.g., Wilkins (1988) and Hammond (1989)), or on the fringes of the field (e.g., Wilensky (1983), and Dean and Wellman (1991)).

The formalisation of planning knowledge is currently an active area of research. Allen, Kautz, Pelavin

---

[24] Suitable URLs are `http://isx.com/pub/ARPI/ARPI-pub/krsl/krsl-info.html` and `http://www.ai.rl.af.mil/PI/Ontology.html` for the Knowledge Representation Specification Language and the Planning Ontology Project, respectively.

and Tenenberg (1991) formalise some existing planning systems, but emphasise plan analysis and plan recognition, rather than plan generation. A few books give individual algorithms, e.g., Ramsay and Barratt (1987) and Thornton and du Boulay (1992). Valente (1994; 1995) summarises the key knowledge representations and algorithms in plan generation using the CommonKADS method.

Overview material on plan generation is more readily found in papers. Steel (1987) gives a straightforward introduction, addressing more advanced planning topics in (Steel, 1988). A comprehensive set of readings in knowledge-based planning has been compiled by Allen, Hendler and Tate (1990). Two key overviews are included: Georgeff (1987) reviews plan generation research, and Tate, Hendler and Drummond (1990) review plan generation techniques.

There are many definitions of plan generation in the above-mentioned literature. Of these, the definition of Tate, Hendler and Drummond (1990, pp. 28-29, italics in original) is most suitable for the purposes of this thesis, because it identifies the key types of knowledge and other data-structures involved in plan generation:

> "An AI planning system is charged with generating a *plan* which is one possible *solution* to a specified *problem*. The plan generated will be composed out of *operator schemata*, provided to the system for each domain of application. ... A problem is characterised by an initial state and goal state description. ... Operator schemata characterise *actions*. ... *Schemata* primarily describe actions in terms of preconditions and effects. ... Each operator schemata characterises a *class* of possible actions, by containing a set of variables which can be replaced by constants to derive operator *instances* that describe specific, individual actions."

From this definition, it can be seen that the inputs to plan generation are:

- A *planning problem*, which itself is characterised by descriptions of the initial and goal states.

- An *operator-set*, i.e., a set of planning operators. A *planning operator* (or an *operator schemata* in Tate, Hendler and Drummond's (1990) terms) is a data-structure which characterises a class of possible actions, either in terms of *preconditions* and *effects* (Tate, Hendler and Drummond, 1990), or in terms of *preconditions*, *add-list*, and *delete-list* (Georgeff, 1987). Planning operators contain *variables*. An *operator instance* is derived by replacing the variables by constants.

Similarly, the output from plan generation is a *plan*, which is a sequence of operator instances (Steel, 1988) representing a course of action (Wilensky, 1983) to solve the planning problem (Tate, Hendler and Drummond, 1990). A plan is a data-structure which must be converted into actions or behaviour in order to change the state of the domain (Nilsson, 1980; Wilensky, 1983; Steel, 1988). The conversion process is generally known as *execution* of the plan. The domain for which plans are generated is *constrained* (Georgeff, 1987).

Tate, Hendler and Drummond's (1990) definition omits four types of domain knowledge that, in my view, are also important in plan generation:

- *Control knowledge*. Control knowledge is vital to the efficiency of plan generation. Other researchers have investigated the learning of control knowledge. As my research concentrates on domain knowledge, this omission is unimportant.

- *Domain objects.* In illustrating the use of the `pickup` operator, Tate, Hendler and Drummond (1990) explicitly mention some domain objects: the object being picked-up, the table and the hand. Moreover, they mention that the MOLGEN (Stefik, 1981) and SIPE (Wilkins, 1988) planners viewed the objects being manipulated as scarce resources on which conflicts occur and need to be avoided. However, their definition fails to identify explicitly domain objects as a type of knowledge. Domain objects are essential to the POI ontology.

- *Inter-object relationships.* Tate, Hendler and Drummond (1990) also omit inter-object relationships. Inter-object relationships (e.g., designation and co-designation) are essential to the POI ontology.

- *Domain constraints.* Nilsson's (1980), Georgeff's (1987), and Steel's (1988) definitions imply that plans are generated in such a way that successful execution is guaranteed. However, successful execution can only be guaranteed at the time of plan generation if the *STRIPS assumption* holds. Informally, the STRIPS assumption is that the initial state is only changed by the set of additions to and deletions from the statements modelling the domain's state (Tate, Hendler and Drummond, 1990). This implies that the only agent capable of altering the domain's state is the plan-executor and that the operators are a correct model of the domain's behaviour. Therefore, a key type of knowledge omitted from Tate, Hendler and Drummond's (1990) definition is domain constraints[25]. Domain constraints are essential to the POI ontology.

In summary, consideration of Tate, Hendler and Drummond's (1990) definition shows that the following types of knowledge need to be represented:

- *Domain models.* Domain models consist of domain objects and inter-object relationships. The representation of domain models is discussed in Section 2.2.3.

- *Domain states and state-change.* The representation of domain states and state-change is discussed in Section 2.2.4.

- *Planning operators.* The representation of planning operators is discussed in Section 2.2.5.

- *Plans.* The representation of plans is discussed in Section 2.2.6.

- *Domain constraints.* The representation of domain constraints is discussed in Section 2.2.7.

2.2.3 Representing Domain Models

Model-based reasoning is well established in AI fields such as diagnosis and simulation. However, it is less commonly found in knowledge-based planning. Closely related knowledge representations that have been used in planning include:

- *Frames.* Minsky's (1975) frame-based representation was the basis of the present-day object-oriented representation. It was an object-attribute-value representation, with classes but lacking inheritance.

---

[25] Tate, Hendler and Drummond (1990) did note that several authors have used constraints, such as resource levels and time constraints on actions, to prune the search space during plan generation.

- *Winston's (1979) representation.* Winston (1979, pps. 162-163) used a representation based on objects (cf. entities), relations (cf. relationships), properties (cf. attributes), and actions.

- *Semantic networks.* Semantic networks (Findler, 1979) can be regarded as the AI equivalent of a pure entity-relationship method. Sparck Jones (1990) identifies two major difficulties exhibited by semantic networks:

  - *Representing any partitioning or grouping of net elements.*

  - *Searching the network.* She states that "network searching conspicuously manifests the general AI problem of the combinatorial explosion" (*ibid.*, p. 121).

- *Conceptual graphs.* Conceptual graphs (Sowa, 1984) address the difficulties exhibited by semantic networks. A *conceptual graph* is a directed graph with two kinds of nodes: concept nodes and relation nodes. *Concepts* are understood as sets of features, and may represent any entity, action, or state that can be described in language. *Relations* specify the roles of concepts and the relationships between concepts. When concepts represent entities/objects, the conceptual graphs are a form of pure entity-relationship model. Sowa acknowledges his debt in the following words (*ibid.*, p. 190, italics in original): "The boxes and diamonds of a [conceptual] graph resemble Chen's *entity-relationship diagrams* (1976), and the similarity is intended." Conceptual graph theory provides a hierarchy of concept-types. This is useful both for partitioning a domain and for speeding the amalgamation of conceptual graphs.

The POI ontology is most closely related to Sowa's (1984) conceptual graph theory. They are similar in that the POI ontology also employs:

- *Explicitly-modelled objects and relationships.*

- *Hierarchies of types.* Both conceptual graph theory and POI support domain-specific type hierarchies, as well as inheritance. Like conceptual graph theory, POI exploits the inheritance hierarchy in reducing combinatorics.

- *Modularised domain knowledge.* In conceptual graph theory, domain knowledge resides in the concept-types; in POI, the knowledge is *owned* by the object-classes.

- *Domain knowledge in the form of heuristics and meta-heuristics.* POI's domain constraints are equivalent to conceptual graph theory's heuristics. Moreover, they follow from the domain structure.

They differ in *typing*. In POI, typing is class-based, taking the form of classes of domain objects and relationships. By contrast, typing in conceptual graph theory is not class-based but based on *prototypes* (Tomlinson, Scheevel and Won, 1989). Sowa concedes (*ibid.*, p. 198-199, italics in original) that "[f]inding the best candidates to match, however, may require extensive searching. ... The danger that all searching programs have to face is the possibility of a *combinatorial explosion*: the amount of calculation can grow exponentially". Tomlinson, Scheevel and Won show that class-based systems have performance advantages over prototype-based systems. It is for this reason that the POI ontology is class-based. To regain performance, Sowa applies heuristics that use background or domain knowledge to eliminate unpromising paths. The heuristics follow from the graph structure. Domain dependencies reside in the concept-types, each of which is a "packet of knowledge about some particular domain" (*ibid.*, p. 201). The procedures that handle them are general rules or *meta-heuristics* that apply to any domain.

A problem could arise with domains which have a potentially infinite number of states. Since such domains might have an infinite number of planning operators, inducing them would require an infinite length of time and/or an infinite computer storage capacity. I avoid this problem by confining the POI ontology to domains with:

- a finite number of classes of domain objects.
- a finite number of binary relationships between any pair of classes.
- a finite number of instances of each object-class.

2.2.4 Representing States and State-Change

The majority of knowledge-based planning research is based on the state-transition model of change. This is most clearly stated by Georgeff (1987, pp. 360-361, italics in original), who writes:

> "The traditional approach has been to consider that, at any given moment, the world is in one of a potentially infinite number of *states* or *situations*. A world state may be viewed as a snapshot of the world at a given instant of time. A sequence of world states is usually called a *behaviour* ...
>
> The world can change its state only by the occurrence of an *event* or *action*. An *event type* is usually modelled as a set of behaviours, representing all possible occurrences of the event in all possible world histories. ... An *event instance* is a particular occurrence of an event type in a particular world history."

There are a number of ontological options relating to the state-transition model:

- *Distinguishing events and actions.* Georgeff (1987) points out that philosophers make much of the distinction between events and actions, i.e., events that are performed by some agent (usually in some intentional way). The POI ontology is only concerned with the more general concept of an event.
- *Distinguishing event and state-change.* In software engineering, state-changes are distinguished from the events which trigger them. However, Georgeff (1987) does not make this distinction. The POI ontology follows the software engineering approach in distinguishing state-change from triggering events.
- *Concurrent state-change.* Georgeff (1987) restricts his attention to domains in which there is no concurrent state-change, as when a single agent acts in a static environment. In such domains, it is only necessary to consider the initial and final states of any action, because nothing can happen between those states. In the POI ontology, concurrent state-change can be excluded by choosing the SA/SSC meta-heuristic. Concurrent action by a single agent can be allowed by choosing the SA/MSC meta-heuristic, and concurrent action by multiple agents can be allowed by choosing the MA/SSC or MA/MSC meta-heuristics.
- *Deterministic state-change.* Georgeff (1987) restricts his analysis further to deterministic events, so that the relation between initial and final states is purely functional. Events may be composed from other events by set union, by intersection and by sequencing. Other researchers who use the same (or an equivalent) state-transition representation include Tate,

Hendler and Drummond (1990) and Steel (1987; 1988). The POI ontology also assumes deterministic state-change.

- *Duration of states and state-changes.* Georgeff (1987) is unclear whether state-change is instantaneous or occurs over an interval of time. In the software engineering field, the state-transition model assumes that state-change occurs instantaneously. Instantaneous state-changes are termed *transitions*. The POI ontology will be based on the state-transition model.

- *Event-instances and event-types.* Georgeff's (1987) definition distinguishes clearly between event-instances and event-types. The POI ontology extends this distinction to states (to give state-instances and state-classes) and to transitions (to give transition-instances and transition-classes). Generalisation from instances to classes is achieved by replacing constants (i.e., names of object-instances) by variables.

One important issue is how to represent states. Options include:

- *Symbolic representation.* Each state could be given a symbolic name, e.g. `State 23`, `block1-held`, `block2-on-block3-and-block1-on-table`, or `block1-state3`. Genesereth and Nilsson (1987) term such symbolic names *state designators*.

- *Attribute-based representation.* Each state could be represented using the object-attribute-value model, e.g., `block1's support = hand1`, `block2's support = table1`.

- *Relationship-based representation.* Each state could be represented using the object-relationship model, e.g., `[holding hand1 block1]` is TRUE, `[holding hand1 block2]` is FALSE, `[on block1 block2]` is FALSE, `[on block2 block2]` is FALSE, `[supporting table1 block1]` is FALSE, and `[supporting table1 block2]` is TRUE. There are various sub-options, such as whether to use prefix, infix (e.g., Thornton and du Boulay (1992)), or postfix notation, and whether to maintain just TRUE relationships or both TRUE and FALSE in the database (e.g., Ramsay and Barratt's (1987) `[not [holding block1]]`).

- *Predicate representation.* Each state could be represented as a predicate, e.g., *ON(B,C)* as in Nilsson (1980).

POI describes states using a relationship-based representation, with:

- a prefix notation, and

- FALSE relationships being maintained in the database as TRUE inverses.

This representation has the following advantages:

- States can be readily generalised by replacing the names of object-instances by variables.

- State descriptions represented the complete domain (*world states*) can be readily partitioned according to the object-classes (*class-states*) or object-instances (*instance-states*) involved.

- The user of the POI is free to choose whether or not to employ the Closed World Assumption (Reiter, 1978).

Another important issue is how to represent transitions. Options include:

- *Symbolic representation.* Each transition could be given a symbolic name, e.g. `transition 32`, `pickup-block1`, or `hand1-transition3`.

- *Attribute-based representation.* Transitions could be represented by providing a `states` attribute whose value was an ordered list of state-identifiers. Alternatively, each transition could have two attributes: `precedingState` and `succeedingState`.

- *Relationship-based representation.* Transitions could be represented by providing a relationship which links pairs of states, such as `[precedes state1 state3]`.

- *Predicate representation.* Transitions could be represented as predicates, e.g., `transition(state1, state2)`.

In the POI ontology, the attribute-based representation is used for transitions. State-instances and classes have `transitions` attributes, whose value is a list of transition-instances or -classes, respectively. Transition-instances and -classes have `states` attributes, whose value is an ordered list of state-instances or -classes, respectively.

In summary, the POI ontology incorporates a state-transition model in which:

- *states* are described using a relationship-based representation, with a prefix notation and with the option of maintaining FALSE relationships in the database as TRUE inverses;

- *transitions* are instantaneous, deterministic, and triggered by events;

- there are *classes* and *instances* of both states and transitions;

- states and transitions are explicitly modelled and have attributes which are used to model the *state-transition network*; and

- *concurrency* can be controlled by suitable choice of the meta-heuristic.

2.2.5 Representing Planning Operators

There are many representations for planning operators, In his review, Georgeff (1987) relates the properties of the domain to world states. This enables him to introduce McCarthy and Hayes' (1969) *situation calculus*, which is based on the notion of a *fluent*, i.e., a function defined on world states. The logical terms of the calculus are used to denote the states, actions, and fluents of the problem domain. Although the situation calculus is highly expressive, a major problem is the large number of axioms needed to describe which properties are unaffected by actions. These are called *frame axioms*. Being forced to specify frame axioms is known as the *frame problem* (Hayes, 1973). Shoham (1987) has attempted to define the frame problem in its most general form.

A variety of alternative logical formalisms to the situation calculus have been developed. There are *modal logics* (Turner, 1984), which avoid the explicit use of terms representing world state. One kind of modal logic, called *temporal logic* (Shoham, 1986), introduces various *temporal operators* for describing properties of world histories. The use of temporal operators corresponds closely with the way tense is used in natural language. *Process logics* (Nishimura, 1980) are another kind of modal

logic which introduce programs (or plans) as additional entities in the domain. *Dynamic logics* (Harel, 1979) can be viewed as a subclass of process logics that are solely concerned with the input-output behaviour of programs. Any of the modal logics could be used to represent domain knowledge instead of the situation calculus (Lansky, 1987; S. Rosenschein, 1981; Stuart, 1985). The expressive power of first-order modal logics would be required for most interesting domains, but suitable theorem provers are currently unavailable (Georgeff, 1987).

Georgeff (1987) turns to one of the most widely used alternatives to the situation calculus, known as the *STRIPS representation*. In the STRIPS representation, a world state is represented by a conjunctive set of logical formulae, actions are represented by STRIPS operators, and inference is performed using the STRIPS rule. The *STRIPS rule* uses the STRIPS operators to determine the descriptions of successive states, as follows:

> "Given a description of a world state *s*, the precondition of an operator is a logical formula that specifies whether or not the corresponding action can be performed in *s*, ... the add-list specifies the set of formulas that are true in the resulting state and must therefore be added to the set of formulas representing *s*, while the delete list specifies the formulas that may no longer be true and must therefore be deleted from the description of *s*. ...
>
> It is important to note that the formulas appearing in the delete list of an operator are not necessarily *false* in the resulting state; rather, the truth value of each of these formulas is considered unknown (unless it can be deduced from other information about the resulting state)." ((Georgeff, 1987), pp. 364 and 365, italics in original.)

Inferencing using the STRIPS rule results in the generation of a linear sequence of instantiated operators, i.e., a linear plan.

There are various extended forms of the STRIPS operator. Tate's (1976) Task Formalism, which forms the basis for the knowledge representations in NONLIN and OPLAN2, enables preconditions and effects to be partially ordered, so that an operator becomes, in effect, a "subproject" network. The operator description language in the SIPE planner (Wilkins, 1988) also allows partially-ordered actions to be specified (as a *plot*). In addition, an *arguments* list provides templates for creating planning variables and adding constraints to them. In particular, variables can be constrained, by names, to be members of certain domain classes. This idea is also used in the POI ontology. Another extension to the STRIPS operator is to permit the specification of source code in the underlying programming language (usually Lisp or Prolog). For example, the IPEM planner (Ambros-Ingerson and Steel, 1988) provides operators with an *executed_by* slot which can be used to specify Prolog code for numeric calculations, for user interface manipulations, or, in its MCS multi-agent systems testbed application (Doran, Carvajal, Choo and Li, 1990), for communication between agents.

As for states and transitions, there are several aspects relating to the representation of planning operators:

- An operator can have pre- and post-conditions, or it can have preconditions-, delete- and add-lists. It is straightforward to convert one representation to the other.

- The logical formulae in the preconditions-, delete- and add-lists (or pre- and post-conditions lists) should be represented in the same way as states. During plan generation, planning operators are sequenced by matching the lists to state descriptions.

- The preconditions may include or exclude filters (Charniak and McDermott, 1985). The difference between preconditions and filters is that preconditions are used to set up subgoals and filters are not. A common application of filters is to check that an object is of the correct class, e.g., `[isBlock ?Block1]` would be a typical filter in the blocks world.

I choose to represent planning operators as STRIPS operators with filter preconditions and with delete- and add-lists. The logical formulae in the preconditions-, delete- and add-lists are generalised relationships, facilitating matching to state descriptions.

2.2.6 Representing Plans

In the knowledge-based planning literature, plans are described as wholly- or partially-ordered sequences of instantiated operators. This is reflected in the definitions of plan generation, including that of Tate, Hendler and Drummond (1990). For example, Nilsson (1980, p. 282) gives as an example the plan consisting of the (wholly-ordered) sequence: {**unstack**(*C*,*A*), **putdown**(*C*), **pickup**(*B*), **stack**(*B*,*C*), **pickup**(*A*), **stack**(*A*,*B*)}. Nilsson also shows this plan graphically. In his Figure 7.3 (reproduced at Figure 11) he shows the plan as a sequence of arcs with thicker lines within the state-space for the three-blocks world.

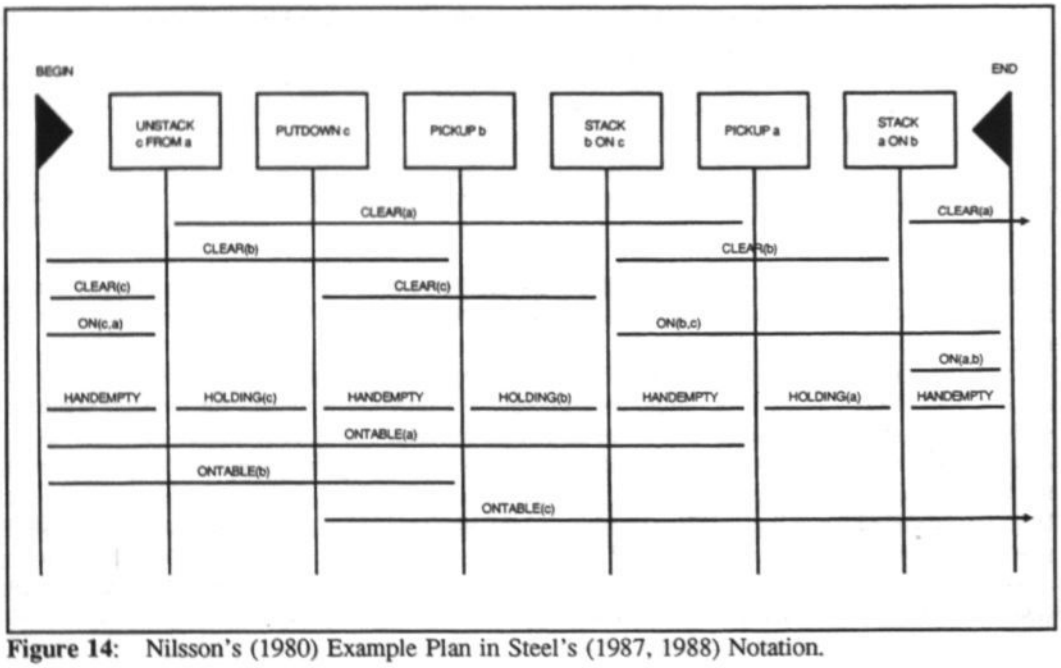

Figure 14: Nilsson's (1980) Example Plan in Steel's (1987, 1988) Notation.

Earlier in this chapter I have observed that Nilsson's (1980) Figure 7.3 is a state-transition network. States are shown as nodes, and transitions are shown as arcs. There are graphic notations in which this mapping is reversed, so that transitions are shown as nodes. For example, in Steel's (1987; 1988) graphic notation an instantiated planning operator is shown as a box with a "tail"[26]. The operator's preconditions are listed to the left of the "tail", and its effects to the right. Plans are shown as a

[26] For this reason, the notation is informally known as Steel's "tadpole" notation.

sequence of operators, linked by arcs from the effects of one operator to the preconditions of a succeeding operator. The arcs represent *protections*, also known as *holding periods*, *ranges*, or (more generally) *goal structure* (Tate, 1975). The initial state is shown as a special "operator", labelled "Begin", with effects but no preconditions. By analogy, the goal state is labelled "End" and has preconditions but no effects. Figure 14 depicts Nilsson's example plan using Steel's notation.

The existence of alternative notations with either states or transitions as nodes indicates that a plan can be described as a sequence of states, a sequence of transitions, or a sequence of alternating states and transitions. Although unusual in knowledge-based planning circles, I choose to represent plans as sequences of world-state descriptions. This leads to the following advantages:

- Plans can be related directly to world- and object-state descriptions.

- Plans can be directly related to their initial and goal state descriptions.

- A close analogy can be drawn between plans and project networks. Operator instances correspond to activities. Steel's (1987; 1988) notation is an enhanced form of the "activity-on-the-node" representation[27]. One operator instance is a direct successor to another when one of its preconditions is linked by a range to one of its predecessor's effects.

- Most importantly, it becomes possible to use the *single-representation trick* (Cohen and Feigenbaum, 1982) in POI's induction step, i.e., the same representation (collections of relationship-instances) can be used both as inputs and as outputs of the induction process. Section 2.3.3 describes the single-representation trick in the machine learning context.

2.2.7 Representing Domain Constraints

I observed above that Tate, Hendler and Drummond's (1990) definition of plan generation omitted to mention domain constraints. By contrast, Georgeff's (1987, p. 367) definition explicitly mentions domain constraints, as follows:

> "Plan synthesis concerns the construction of some plan of action for one or more agents to achieve some specified goal or goals, given the constraints of the world in which these agents are operating."

Plan generation is usually viewed as a search problem (Korf, 1987). The search paradigm of problem-solving (Simon, 1983b) depends on the existence of a domain-specific problem space. There are two leading problem spaces in plan generation (Tate, Hendler and Drummond, 1990):

- *State space*. Planners prior to 1975 defined points in their search space as descriptions of domain states. The domain's state-space is traversed by applying an applicable planning operator to move from one state to another.

- *Plan space*. After 1975, planners defined points in their search space as partially-elaborated plans. The domain's plan-space is traversed by applying an applicable planning transformation, such as expanding an action to a greater level of detail.

---

[27] It is enhanced in that dependencies are detailed as ranges.

Correspondingly, domain constraints can be applied for plan generation purposes in state-space or plan-space. State-space domain constraints are more frequently found in the knowledge-based planning literature. Drummond (1987) used state-based domain constraints to reason about the effects of actions, with the plan generation and execution systems using the same reasoning mechanism: *domain constraint reconciliation*. Drummond and Currie (1989) called domain constraints *invariants*, and Rich and Knight (1991) called them *logical statements*. Tenenberg (1991) employed what he called *static axioms*, defined as axioms that are true in every state, to ensure that only valid state descriptions (which he called *legal situations*) were used in plan generation. Bresina, Drummond and Kedar (1993) identify two roles for state-based domain constraints:

- to maintain internal model consistency, either predictively or reactively, by describing the physics of the domain in terms of impossibilities. They termed such constraints *domain constraints*. The representation and reasoning mechanisms from Drummond (1987) and in Drummond and Currie (1989) were employed.

- to define goals as conditions that must be maintained or prevented over an interval of time, rather than just in terms of a desired world state. They termed such constraints *behavioural constraints*. Behavioural constraints were represented as predicates.

Plan-space domain constraints were introduced by Stefik (1981) in his MOLGEN planning system to resolve interactions by ordering operator-instances using a *constraint-posting* technique. Other planning systems which used similar constraint-posting techniques in plan-space include Chapman's (1987) TWEAK and Dean, Firby and Miller's (1988) FORBIN.

Whether applied in state- or plan-space, domain constraints are used for pruning the search space[28]. The POI ontology uses domain constraints in the same role, but differs in that:

- The search space being pruned is the domain's *version space*, rather than its state space or plan space.

- Pruning occurs during induction, rather than during plan generation. In the context of version spaces, pruning is known as *candidate elimination*.

Version space and candidate elimination are described in Section 2.3.

$$[\exists x\colon \mathrm{HOLDING}(x)] \rightarrow \neg \mathrm{ARMEMPTY}$$
$$\forall x\colon \mathrm{ONTABLE}(x) \rightarrow \neg \exists y\colon \mathrm{ON}(x,y)$$
$$\forall x\colon [\neg \exists y\colon \mathrm{ON}(y,x)] \rightarrow \mathrm{CLEAR}(x)$$

Figure 15: Rich and Knight's (1991) Logical Statements for Blocks World.

In the plan generation literature, domain constraints generally appear in the form of production rules. For example, Figure 15 reproduces Rich and Knight's (1991) logical statements for the blocks world. In Rich and Knight's words (*ibid.*, p. 333):

[28] Viewed in terms of the Generate-and-Test paradigm, constraints are used in the Test phase. Constraints are used in the Generate phase in the related field of knowledge-based scheduling, where the problem-solving paradigm (Simon, 1983) is constraint resolution.

"The first of these statements says simply that if the arm is holding anything, then it is not empty. The second says that if a block is on the table, then it is not also on another block. The third says that any block with no blocks on it is clear."

```
Holding(x) ∧ y ≠ x ⊃ ~Holding(y)                    (1)
HandEmpty ⊃ ~Holding(x)                            (2)
~On(x, x)                                          (3)
On(x, y) ⊃ ~Clear(y)                               (4)
On(x, y) ⊃ ~On(y, x)                               (5)
On(x, y) ⊃ Above(x, y)                             (6)
Above(x, y) ∧ Above(y, z) ⊃ Above(x, z)            (7)
```

Figure 16: Tenenberg's (1991, p. 222) Blocks-World Static Axioms.

Tenenberg's (1991) blocks-world static axioms are also expressed in production-rule form; see Figure 16. By contrast, domain constraints in Drummond (1987), Drummond and Currie (1989), and Bresina, Drummond and Kedar (1993) are expressed as tuples. Bresina, Drummond and Kedar's behavioural constraints are expressed as predicates. Drummond (1987, p. 200) gives example tuples for the blocks world (see Figure 17), and describes them as follows:

"The first of these [tuples] says that it is impossible for the agent to believe there is any block y_1 which is both clear and under some other object. The second says that it is impossible for the agent to believe that there is any block x_2 which is on top of two different objects simultaneously. The third domain constraint specifies that it is impossible to believe that there is any block y_3 which is under two different blocks. The fourth says that it is impossible for the agent to believe that any given object x_4 is both a block and a table. The final constraint specifies that it is impossible to believe that one object has more than one weight."

```
{{on(x1,y1), clear(y1), block(y1) },
 {on(x2,y2), on(x2,z2), block(x2) },
 {on(x3,y3), on(z3,y3), block(y3) },
 {block(x4), table(x4) },
 {weight(x5,x6), weight(x5,x7) }}
```

Figure 17: Drummond's (1987, p. 200) Domain Constraints.

Like Tenenberg (1991) and Rich and Knight (1991), the POI ontology also uses the production-rule formalism. However, the rules express exclusion constraints (Nijssen and Halpin, 1989) between pairs of relationships. An *exclusion constraint* states that, in all world states, the pair of relationships are mutually exclusive. As with Tenenberg's static axioms, exclusion constraints are true in every world state. Moreover, like Drummond's (1987) and Bresina, Drummond and Kedar's (1993) domain constraints and Drummond and Currie's (1989) invariants, exclusion constraints express impossibilities. However, exclusion constraints are stronger in that they express real impossibilities in the domain, not just in an agent's beliefs about that domain.

Figure 18 shows Rich and Knight's (1991) logical statements expressed as exclusion constraints. For convenience, exclusion constraints are implemented as *exclusion-rules* of the form:

```
IF <antecedent1> AND <antecedent2> THEN INVALID,
```

where an exclusion rule always has:

- one or more antecedents, each of which matches a subset of relationship-instances; and
- the single consequent INVALID, which expresses impossibility.

```
⊬ [∃x: HOLDING(x)] ∧ ARMEMPTY
⊬ ∀x: [ONTABLE(x) ∧ [∃y: ON(x,y)]]
⊬ ∀x: [CLEAR(x) ∧ [∃y: ON(y,x)]]
```

Figure 18: Rich and Knight's (1991) Logical Statements.

Figure 19 shows the exclusion-rules that would correspond to Tenenberg's (1991) static axioms. Axioms (6) and (7) have no equivalent in the POI ontology because above would be a second-order relationship. Axiom (7) would also require an exclusion-rule with three antecedents.

```
IF holding ?Hand1 ?Block1 AND holding ?Hand1 ?Block2 THEN
INVALID.                                                    (1)
IF holding ?Hand1 NIL AND holding ?Hand1 ?Block1 THEN INVALID.(2)
IF on ?Block1 ?Block1 THEN INVALID.                         (3)
IF on ?Block1 ?Block2 AND on NIL ?Block2 THEN INVALID.      (4)
IF on ?Block1 ?Block2 AND on ?Block2 ?Block1 THEN INVALID.  (5)
```

Figure 19: Exclusion-Rules for Tenenberg's (1991) Axioms.

2.3 MACHINE LEARNING

This section builds on the preceding two sections by reviewing the machine learning techniques applicable to the learning of planning operators. Since systems that learn domain knowledge tend to rely on induction (Minton and Zweben, 1993), the review focuses on inductive learning, and, in particular, on learning from examples.

First, machine learning and induction are defined. Second, a simple model of the design of a learning system is outlined. Third, issues relating to learning from examples are reviewed. Fourth, Mitchell's (1982) version space and candidate elimination algorithm and its application of Mitchell's algorithm to POI are described. Fifth, the POI algorithm is compared with other approaches to operator learning. Finally, POI is placed into the wider context of all machine learning techniques.

2.3.1 Defining Machine Learning and Induction

The machine learning literature is large. Readers interested in an overview of the field are advised to consult the readings on machine learning (Shavlik and Dietterich, 1990) and on knowledge acquisition and machine learning (Buchanan and Wilkins, 1993). Other reviews of machine learning may be found in volume 3 of the Handbook of Artificial Intelligence (Cohen and Feigenbaum, 1982), in a special issue of the Artificial Intelligence journal (volume 40, numbers 1 to 3), Carbonell (1989), and in Kocabas (1991). There is also an influential series of books entitled "Machine Learning: The AI approach", of which four volumes have been published to date. There are several introductions to machine learning; (Kodratoff, 1988) is technical and aimed at postgraduates, (Partridge and Paap,

1988) is idiosyncratic, and (Forsyth, 1989) is biassed towards biologically-inspired systems. In addition to the general AI journals, there are several that specialise in machine learning; the Machine Learning Journal is one of the most notable. Additional material can be found in the proceedings of the workshops and international conferences devoted to machine learning.

There are various definitions of learning in the Machine Learning literature. In the early days of Artificial Intelligence, McCarthy (1958) argued that a prerequisite for machine learning is the ability to represent knowledge in structures that can be modified when the learning system is told new things. Simon (1983b, p. 28, italics in original) added the requirement that the modifications should be adaptive:

> *"Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time."*

Michalski (1993, p. 7) identified the goal-directedness of the adaptive process, as follows:

> "Any learning process can be viewed as a self-modification of the learner's current knowledge through an interaction with some information source. Such knowledge modification is guided by the learner's desire to achieve a certain outcome, and can engage any kind of inference."

Simon's (1983b) definition is most widely accepted in the Machine Learning community. However, Partridge and Paap (1988) have challenged Simon's definition on the following grounds:

- It emphasises the beneficial effects of learning, but prejudices and bad habits are also learned.
- It assumes improvement is monotonic.
- It limits learning to similar tasks and tasks drawn from the same population.

The first two criticisms are irrelevant to my research. The POI algorithm makes no value judgements about the input information it is given. If the input information is incorrect or incomplete, then the induced operator-set will necessarily also be incorrect or incomplete. Non-monotonic improvement is important in cognitive science and psychology. However, as I take the engineering approach, my research is not concerned with reproducing the non-monotonicity of human or animal learning.

By contrast, Partridge and Paap's (1988) third criticism is important in my research. To induce planning operators, knowledge transfer between tasks drawn from the same population is not enough. The benefits of learning would be severely limited if the planning operators could only be used for plan generation in the same domain as used to obtain the POI input. Knowledge should be transferable at least to domains with the same object-classes and relationship-classes but with differing numbers of object-instances. Better still, knowledge should be transferable between domain variants or from a domain to its specialisation, e.g., from the blocks world to the domain of container handling.

Carbonell (1989) has extended Simon's (1983b) definition to include the ability to perform new tasks, without requiring that they be drawn from the same population. This meets my need to be able to transfer knowledge between domains. Carbonell's definition is as follows (*ibid.*, p. 2):

> "... learning can be defined operationally to mean the ability to perform new tasks

that could not be performed before or [to] perform old tasks better (faster, more accurately, etc) as a result of changes produced by the learning process."

Inherent in Carbonell's (1989) definition is a distinction between two types of machine learning:

- *The system can perform new tasks.* This is "usually called *empirical learning* or *inductive learning*, since it is typically accomplished by reasoning from externally supplied examples to produce general rules or procedures" (Shavlik and Dietterich, 1990, p. 1, italics in original).

- *The system can perform old tasks better.* This is "often called *speedup learning* or *skill acquisition*" (Shavlik and Dietterich, 1990, p. 2, italics in original). Speedup learning is often based on deductive reasoning. Shavlik and Dietterich (1990) list two ways to speed up search in problem-solving systems: to introduce macro-operators, and to introduce meta-level control knowledge. Buchanan and Wilkins (1993) identify a third way: compilation from deep domain models.

This thesis is concerned solely with inductive learning, and, in particular, with learning from examples. I adopt Carbonell's (1989) definition of learning and Shavlik and Dietterich's (1990) definition of induction.

None of the definitions quoted above consider the nature of the tasks to be performed. This can affect the type of information to be learned and the learning process. Dietterich (1986) distinguished symbol-level and knowledge-level learning. In POI, the task to be performed is plan generation, which is itself a knowledge-based inference process. The information to be learned from examples in POI must be at the knowledge level.

More specifically, it is necessary to clarify what are - in Shavlik and Dietterich's (1990) terms - the "examples" and the "general rules and procedures". In the full POI algorithm, the "examples" are the state descriptions that are input to Step (1.1). To correspond to the input state-descriptions, the "general rules or procedures" should - strictly speaking - be state-classes. State-classes are indeed extracted in Step (2.5), but that Step then goes on to extract the corresponding transition-classes. In Step (2.6) the transition-classes are re-formatted as planning operators. Planning operators are clearly at the knowledge level, can be represented using rule-like formalisms[29], and have a procedural reading. Extraction of transition-classes and reformatting them as planning operators can be regarded as post-processing of the induced state-classes. This thesis takes the less strict viewpoint that the induced operators can be regarded as the "general rules and procedures" of the input state description "examples".

2.3.2 Learning System Design

Based on Simon's (1983b) definition of machine learning, Cohen and Feigenbaum's (1982) contributors developed a simple model of the design of a learning system, as shown in Figure 20. The circles denote declarative bodies of information, while the boxes denote functional processes. The arrows show the predominant direction of information flow. The *environment* supplies some information to the *learning element*, the learning element uses this information to make improvements

[29] For example, compare the STRIPS formalism to expert system rules.

in an explicit *knowledge base*, and the *performance element*[30] uses the knowledge base to perform its task. Information gained during attempts to perform the task can serve as feedback to the learning element.

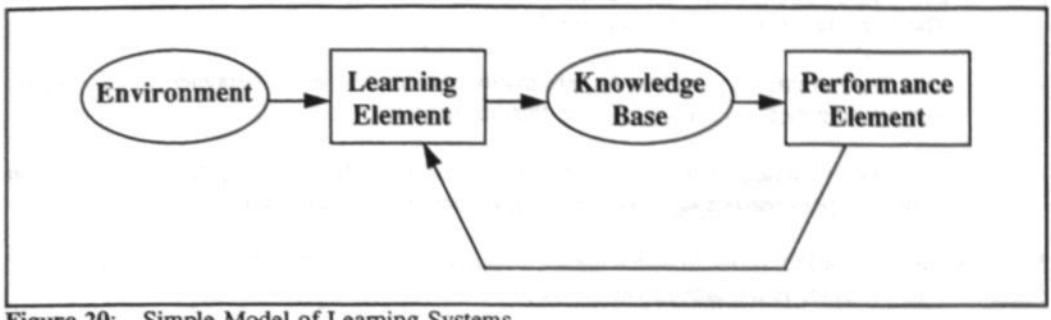

Figure 20: Simple Model of Learning Systems.

In the context of POI (see Figure 21), the environment is the domain whose planning operators are to be induced. The learning element is the POI algorithm, and the performance element is a plan-generation algorithm together with a plan executor. The knowledge base contains the induced planning operators.

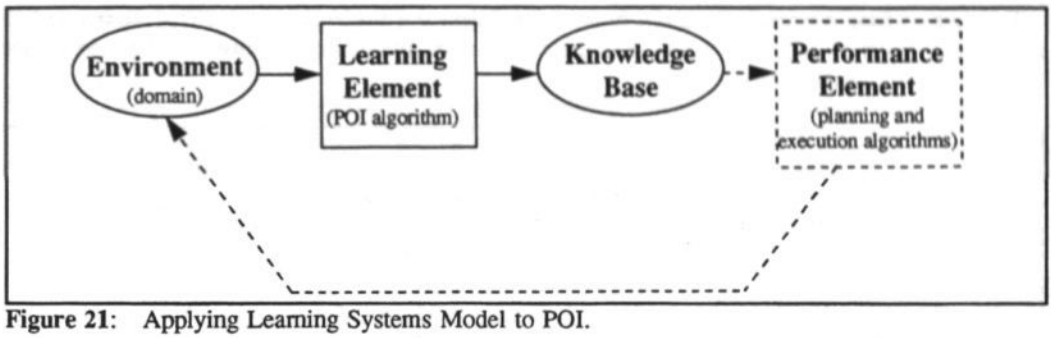

Figure 21: Applying Learning Systems Model to POI.

My research has emphasised POI as an open-loop learning element, with state descriptions or domain model information being input from the environment and the induced operator-set being output. This open loop is shown in Figure 21 by the circles, boxes, and arrows in full lines. For completeness, some POI experiments have been done in which the loop has been closed by coupling POI to a STRIPS planner, a plan executor, and a domain simulation. The performance element feeds information back, not directly to POI (as Cohen and Feigenbaum's (1982) figure suggests), but by changing the state of the (simulated) domain. The additional information flows and processes needed to close the loop are shown by dashed lines.

The choice of learning technique for the POI algorithm can be justified using Cohen and Feigenbaum's (1982) figure. There are four basic learning situations, depending on the relative levels of abstraction between the information provided by the environment and the information needed by the performance task. The appropriate technique is:

---

[30] Some authors refer to the performance element as the *problem solver*. Strictly speaking, this is only applicable when the task to be performed is problem solving, as is the case in planning.

- *Rote learning* when the information provided by the environment is at the same abstraction level as the information needed by the performance task.

- *Learning by being told* when the information provided by the environment is too abstract. The learning element must specialise this information.

- *Learning from examples* when the information provided by the environment is too specific. The learning element must generalise this information.

- *Learning by analogy* when the information provided by the environment relates only to an analogous performance task. The learning element must analogise this information.

Since the situation for POI is one in which the information supplied by the environment is too specific, learning from examples is the appropriate technique.

2.3.3 Two-Space Model for Learning from Examples

Simon and Lea (1974) view the problem of learning from examples using a *two-space* model; see Figure 22. The examples - usually known as *training instances* - are drawn from a larger space of all possible instances, i.e., from the *instance space*. Similarly, the rules that the learning program identifies will be drawn from a larger *rule space*. In terms of Cohen and Feigenbaum's (1982) learning-system model, the instance and rule spaces are the information spaces of the environment and of the knowledge base, respectively.

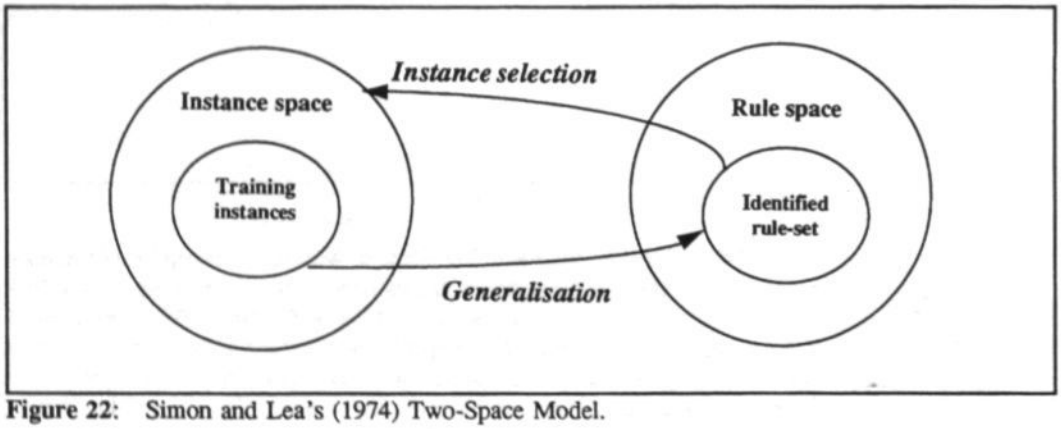

Figure 22: Simon and Lea's (1974) Two-Space Model.

The learning element performs a coordinated search of the instance and rule spaces. Coordination is achieved by means of *instance selection* and *generalisation* processes. As the instances and rules are generally represented using different formalisms, the raw training instances will usually have to be interpreted so that they can guide the search of the rule space. Similarly, instance selection will usually have to include experiment-planning routines to guide the search of the instance space.

Implicit in Simon and Lea's (1974) two-space model is the assumption that both spaces are known in advance. Instances may be selected by a teacher, by the learning element itself, or by the environment (Michalski, Carbonell and Mitchell, 1983). By contrast, in POI neither space is known in advance. Instance selection is assumed to be done by the environment, and, unlike many other operator-learning systems, there is no experiment planning. POI must construct its rule space as

training instances are presented to it, i.e., generalisation in POI is data-driven and incremental. Figure 23 shows in dashed lines the spaces and processes external to POI.

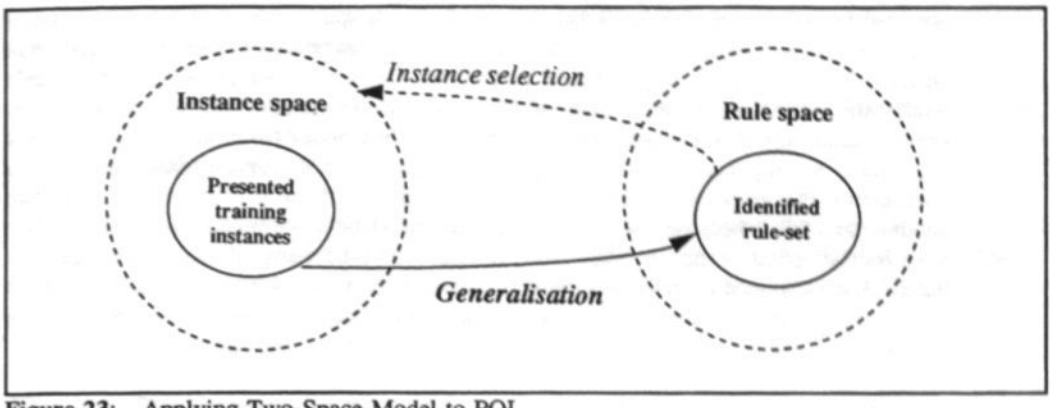

Figure 23: Applying Two-Space Model to POI.

2.3.4 Issues Relating to Learning from Examples

Cohen and Feigenbaum (1982) identify several design issues relating to learning from examples. The design issues relevant to POI are as follows:

- *The quality of the training instances.* Low-quality training instances would invite multiple, conflicting interpretations during generalisation, resulting in ambiguity. Cohen and Feigenbaum (1982) list three sources of ambiguity:
  - *Instances may contain errors.* POI avoids this by assuming that each training instance is complete and correct, e.g., an observation made by perfect sensors.
  - *Instances may be unclassified.* POI avoids this by assuming that each training instance is implicitly classified as a *positive* instance, e.g., an observation of the real-world domain.
  - *The order of presenting the training instances may be poor.* POI avoids this by assuming that there is no information to be derived from the presentation order. Moreover, POI has no way of knowing whether all the training instances have been presented, i.e., when the presentation sequence is complete.
- *The difficulty of interpretation.* Interpretation is difficult when different representations are used for the instance and rule spaces, and this is further compounded when the raw training instances are presented in a sub-symbolic form. POI avoids the first difficulty by employing the *single-representation trick* (Cohen and Feigenbaum, 1982), i.e., it uses the same representation for both spaces. Training instances are viewed as highly specific pieces of acquired knowledge. The second difficulty is avoided by assuming that perception, recognition of objects and relationships, and conversion of the recognised information into a suitable symbolic form are all done outside POI.
- *The form of the rule space.* The rule space representation is influenced by the kind of inference to be supported in the performance element. In the case of POI, the performance

element performs plan generation and execution. Unlike other operator-learning systems which represent the rule space directly as planning operators, POI represents the rule space as domain states, each described as a set of inter-object relationships. Planning operators are obtained by post-processing the induced states. The rule space representation must support generalisation, i.e., inferencing in which one description can be said to be *more general than* another. This is readily implemented in POI by checking whether or not one state's relationship-set is a proper subset of the other state's relationship-set. Furthermore, generalisation should be accomplished by inexpensive, syntactic operations, such as dropping conditions, turning constants into variables, adding options, curve-fitting, and zeroing coefficients. POI employs the first two of these operations. If one state is more general than another, then this is because one or more relationships found in the second state's description have been dropped in the first state's description. Post-processing of the induced states to obtain planning operators involves turning (object-)constants into (object-)variables to extract state-classes from the induced states. Michalski (1980) used sets and set membership in a similar way (which he termed *internal disjunction*) to express generalisation.

- *The methods used to search the rule space.* Cohen and Feigenbaum (1982) distinguish methods in which the presentation of the training instances drives the search (i.e., *data-driven methods*) from those methods in which an *a priori* model guides search (i.e., *model-driven methods*). POI must handle each training instance as it is presented, i.e., it must learn incrementally. Only data-driven methods support incremental learning. Of the two data-driven methods, the *version-space method* (Mitchell, 1982) is the more appropriate because it exploits the single-representation trick. The disadvantage of data-driven methods - that they have poor immunity to noise in the training instances - has already been countered by requiring training instances to be complete and correct.

- *The ability to cater for new terms in the rule space.* In many learning problems, the learning element must confront the possibility that the rule space will expand during generalisation. This causes no difficulty for POI because it maintains a representation only of the identified rule-set. Since, by assumption, all previous training instances are positive, complete and correct, none of them is invalidated by new terms. A new term would appear in the next training instance, either as a new object-instance or as a new relationship. By virtue of the single-representation trick, no interpretation is needed before the new term is generalised to appear in the identified rule-set.

Another important design issue is whether the set of examples includes both positive and negative training instances or is restricted solely to positive instances. In a fundamental paper, Gold (1967) proved that, if a program is given an infinite sequence of positive instances, then the program cannot determine the correct concept in any finite time. Despite Gold's proof, learning from positive instances is observed in real life, e.g., linguists have observed that children learn their native languages from what appears to be positive rather than positive and negative data (Angluin and Smith, 1983). Moreover, Vere's (1978) *maximal unifying generalisation* and Hayes-Roth and McDermott's (1978) *interference matching* learn using only positive training instances. Researchers have made various suggestions for avoiding the difficulty introduced by Gold's proof (e.g., see Angluin and Smith (1983), chapter 4, and Cohen and Feigenbaum (1982), Section XIV, Article D5e). The suggestions all depend on providing additional information to constrain the learning program's choices. The solution employed in POI is to adopt a default rule that all relationship-pairs are forbidden. The default is overridden if a particular relationship-pair has been observed in one or more positive training instances. This solution is closely related to the suggestion to use a prior probability. The default rule provides the additional, negative information needed to supplement the exclusively-positive information obtained from the training instances.

2.3.5 Classifying Symbolic Systems that Learn from Examples

Cohen and Feigenbaum (1982) classify systems that use symbolic representations for learning from examples according to the complexity of the learning task:

- *Learning single concepts.* The simplest task is to classify new training instances according to whether or not they are instances of a single concept. This is the best understood task in machine learning research.

- *Learning multiple, independent concepts.* Many tasks involve the learning of a set of concepts that operate independently. Cohen and Feigenbaum note that this has received some research attention, e.g., in AM (Lenat, 1976), Meta-Dendral (Buchanan and Mitchell, 1978), and AQ11 (Michalski and Larson, 1978).

- *Learning to perform multiple-step tasks.* Cohen and Feigenbaum state that the most complex learning problems are planning tasks that require the performance element to apply a sequence of operators. The identified rules must be chained together into a sequence. Difficult problems of integration and credit-assignment must be confronted. Successful multiple-step learning systems include Samuel's (1959) Checkers Player, Waterman's (1970) Poker Player, Sussman's (1975) HACKER, and Mitchell, Utgoff, and Banerji's (1983) LEX systems.

Cohen and Feigenbaum (1982) observe that the term *concept* is used loosely in the machine learning literature. They define a concept as a predicate, expressed in a description language, that is TRUE when applied to a positive instance of the concept and FALSE when applied to a negative instance. The single-concept learning problem is then stated formally as:

Given:

(1) A description language for concepts, and
(2) A set of (positive and/or negative) training instances,

Find:

The unique concept in the rule space that best covers all of the positive instances and none of the negative instances.

The two standard assumptions (Cohen and Feigenbaum, 1982) are that:

- *The training instances are all examples (or counterexamples) of a single concept.* When this assumption is relaxed to an unordered set of concepts, then the problem becomes one of multiple-concept learning. When the set of concepts is ordered, then the problem is a multiple-step learning one.

- *The single concept can be represented by a point in the rule space.* If this assumption is violated, then a better rule space must be found.

2.3.6 POI as System that Learns from Examples

POI specialises the description language and the standard assumptions in the following ways:

- A concept is taken to be a conjunction of predicates, expressed in a description language.
- Each predicate is TRUE when applied to a positive training instance of that concept, and FALSE in all other situations. Note that FALSE may indicate that the truth value of the predicate is unknown.
- Predicates are either relationships between pairs of object-instances (known as *primary* relationships), or *inverses* of a primary relationship. When an inverse relationship is TRUE, this denotes that the corresponding primary relationship is known to be FALSE for the object-instance named in the inverse relationship.
- The description language is constrained by exclusions between pairs of relationships. The set of exclusion-constraints includes, as a minimum, all pairings of primary relationships and their inverses.
- The training instances are assumed to be examples of a *set* of concepts, rather than a unique concept. In other words, POI is concerned with multiple-concept learning.
- Each concept can be represented by a point in the rule space.

At first sight, it would appear that POI should be classified as learning to perform multiple-step tasks. However, deeper consideration shows that the "concepts" being induced in POI - domain states - are independent from one another. By inducing states rather than operators, POI simplifies a multiple-step learning task into a simpler, multiple-concept learning task. During post-processing, the learned concepts are chained together into sequences using the meta-heuristic.

Several multiple-concept learning systems have been constructed, using a variety of techniques. The techniques used in systems such as Meta-Dendral and AM are inappropriate for POI because they depend on having substantial bodies of initial domain knowledge, in the form of a behavioural model of the domain, e.g., a set of primitive operators. By contrast, POI takes its cue from AQ11 (Michalski and Larson, 1978) which uses an algorithm that is "nearly equivalent to the repeated application of the candidate-elimination algorithm" (Cohen and Feigenbaum, 1982, p. 424). The version space and candidate elimination algorithm has also been used by de Kleer and Williams (1987) to induce multiple concepts. Other researchers' success in adapting the version space and candidate elimination algorithm for multiple-concept learning gave encouragement for its application in POI.

2.3.7 Version Space and Candidate Elimination Algorithm

The version space and candidate elimination algorithm was developed by Mitchell in the late 1970s for single-concept learning purposes. The term *version space* refers to the knowledge representation used, and the term *candidate elimination* refers to the algorithm that manipulates the version space. Mitchell's (1982) paper in the Artificial Intelligence journal will be taken as the primary source, although Mitchell had already presented relevant papers at the 1977 and 1979 IJCAI conferences and his PhD thesis appeared in 1978.

Mitchell (1982) noted that the sentences in any representation language could be partially ordered

according to the generality of each sentence. Syntactic rules of generalisation, such as dropping conditions, can be used to generate the partial order. For example, the sentence:

$$ON(a, b)$$

is a more general description of a particular blocks world situation than is the sentence:

$$ON(a, b) \wedge CLEAR(a)$$

because the latter sentence constrains the top of block *a* to being clear of all other blocks.

By virtue of the partial ordering, the sentences can be seen as elements in a lattice. The bottom element of the lattice consists of the null description, and represents the most general description, i.e., the description from which all conditions have been dropped. The top element consists of the union of all the statements in the representation language, and represents the most specific description, i.e., the description from which no conditions have been dropped. From the single-representation trick, the lattice is both the instance space and the rule space.

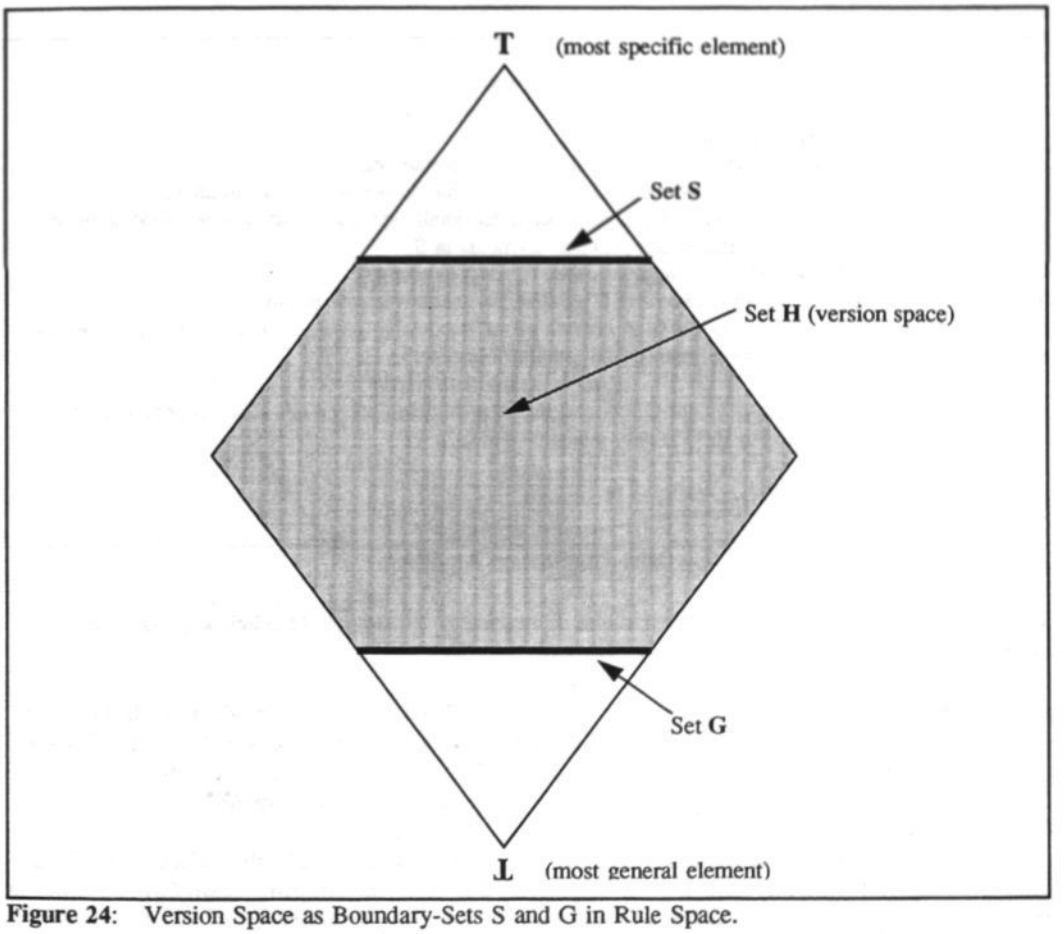

Figure 24: Version Space as Boundary-Sets S and G in Rule Space.

In any real-world domain the rule space is too large to be represented exhaustively. With boolean-

valued predicates, the size of the rule space would be two raised to the power of the number of predicates in the representation language. Memory requirements can be reduced by representing only that part of the rule space which is consistent with the training instances: the *version space*, also known as the *set H of plausible hypotheses*. A *plausible hypothesis* is any concept that has not yet been ruled out by the training instances. Mitchell (1982) pointed out that the version space can be more efficiently represented by the two boundary-subsets G and S (see Figure 24), where G is the subset of the most general elements of set H, and S is the subset of most specific elements of set H. The rules of generalisation must be used to fill in the subspace between G and S whenever the full set H is needed.

Mitchell's (1982) *candidate-elimination algorithm*, detailed in Figure 25, takes advantage of the boundary-set representation of the version space. Initially, the version space is the complete rule space. As training instances are presented, candidate concepts are eliminated by shrinking the version space. Positive instances force the algorithm to generalise by removing over-specific concepts by lowering set S towards the bottom element. Negative instances force the algorithm to specialise by raising the set G towards the top element. In its single-concept application, the candidate-elimination algorithm terminates when the version space has been reduced to a singleton set. The remaining candidate is the desired concept.

Step 1.	Initialise H to be the whole rule space.
Step 2.	Accept a new training instance. If new instance is positive, then (Update-S routine): - remove from G all concepts that do not cover new instance, - update S to contain all of maximally specific common generalisation of new instance and previous elements in S. If new instance is negative, then (Update-G routine): - remove from S all concepts that cover new instance, - update G to contain all of maximally general common specialisations of new instance and previous elements in G.
Step 3.	Repeat Step 2 until G = S and this is a singleton set. (When this has occurred, H has collapsed to include only a single concept.)
Step 4.	Output H (i.e., either G or S).

Figure 25: Mitchell's (1982) Candidate-Elimination Algorithm.

There are a number of factors which make it necessary to modify Mitchell's (1982) candidate-elimination algorithm for application in POI, as follows:

- *Termination cannot be based on reducing the version space to a singleton set.* If the version space does reduce to a singleton set, then the domain's state-space would consist of just one state (i.e., the single learned "concept"). With only one state, state-change would be impossible. Without state-change, there would be no planning operators.

- *There are no negative training instances.* Section 2.3.4 has already shown how this difficulty is solved in POI by adopting the default rule that all relationships-pairs are forbidden, unless the particular relationship-pair is observed in the training instances.

The effect of the first factor is that the result of induction is a subspace, instead of a single point in

rule space. Induction terminates when this subspace cannot be reduced any further. The effect of the second factor is to that set G cannot be distinguished from its initial value: the bottom element. The combined effect is that the version space in POI is bounded by the set S and the bottom element. As in maximal unifying generalisation and interference matching, only set S moves when training instances are presented.

The induction step in the POI algorithm also employs Mitchell's (1982) boundary-set representation of the version space, but with the bottom element replacing set G. Initially, the version space is the bottom element. As each training instance is presented, the predicates are extracted, generalised, re-specialised for other object-instances, and added to the description language. Exclusion constraints relating to newly-added predicates are generated using the default rule. Pre-existing exclusion constraints are updated. The version space is augmented by adding the newly-added predicates to the first level ("ring") above the bottom element. The version space is expanded by beam search upwards until terminated by the current exclusion constraints. Version space search in POI is depicted in Figure 26.

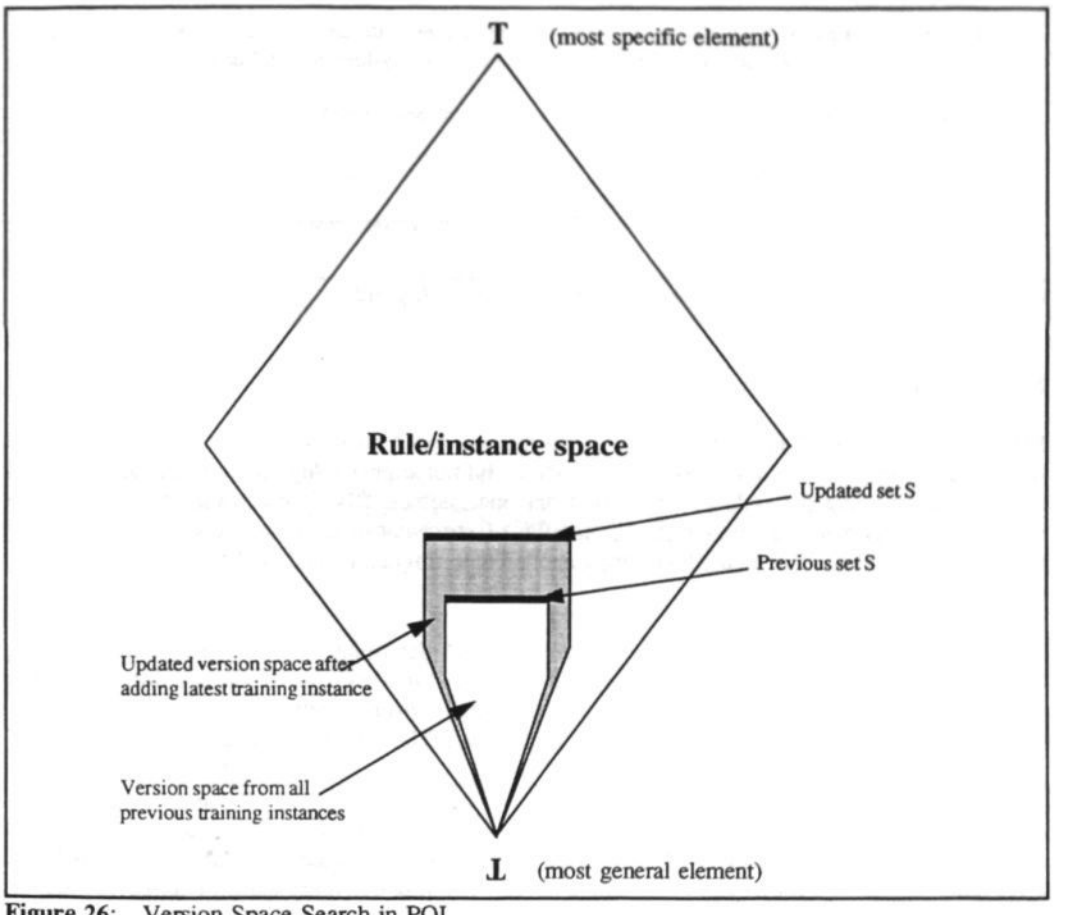

Figure 26: Version Space Search in POI.

A similar scheme, but replacing set S by a single element, has been proposed by Bundy, Silver and Plummer (1985). They noted that the set S never holds more than one concept if the description language is purely conjunctive and attribute-based (Haussler, 1989), with attributes that are tree-

structured. However, Haussler pointed out that the set G can then grow exponentially with the number of training instances.

Up to this point, POI's induction step has been described in its incremental form. For simplicity, the POI implementations employ a one-shot version. The as-implemented induction step, Step (2.5), is detailed in Figure 27.

Step 2.5.1.	Extract description language D from training instances.
Step 2.5.2.	Generate no-goods for each statement in D against all other statements in D, using exclusion constraints.
Step 2.5.3.	Initialise H to be bottom (null) element.
Step 2.5.4.	Initialise S as set of singleton elements, one for each statement in D. Add S to H.
Step 2.5.5.	Remove each element in S and form all unions with statements in D that neither violate no-goods nor already exist in H. Add each newly-formed valid union to S and H.
Step 2.5.6.	Repeat Step 2.5.5 until no new valid unions are formed.
Step 2.5.7.	Output resulting S.

Note: Steps 2.5.5 and 2.5.6 implement the beam search of the version space.

Figure 27: Induction Step (one-shot version) of the POI Algorithm.

2.3.8 Version Space Merging

Hirsh (1989) developed an incremental learning method based on version-space intersection, which he named *incremental version-space merging*. Hirsh did not stipulate any requirements as to how the version spaces being merged should differ from one another. His purpose was to generalise the version-space approach to remove its assumption of strict consistency with data so as to handle uncertain information. The method was implemented in a program named "IVSM", and demonstrated on four different learning tasks.

To use version-space merging in POI, it is necessary to determine how the version spaces being merged should differ from one another. For this purpose, POI adopts two of the solutions commonly found in software engineering for representing large state-transition networks. Section 2.1.3 showed that state-transition networks can be partitioned into sub-STNs. Where partitioning is done by object-classes the sub-STNs are *class-STNs*.

As there would be a sub-version-space corresponding to each sub-STN, version-space merging can be exploited to regenerate the global-STN from the set of sub-version-spaces, i.e., from the class-STNs. In practice, POI does not represent the class-STNs, but just the class-states. This is because the (global) transitions are extracted from the (global) states in POI only after the merging process has been completed. I term the process of regenerating the global-STN from the set of class-states as *class-state merging*. The class-state merging process is shown in Figure 28.

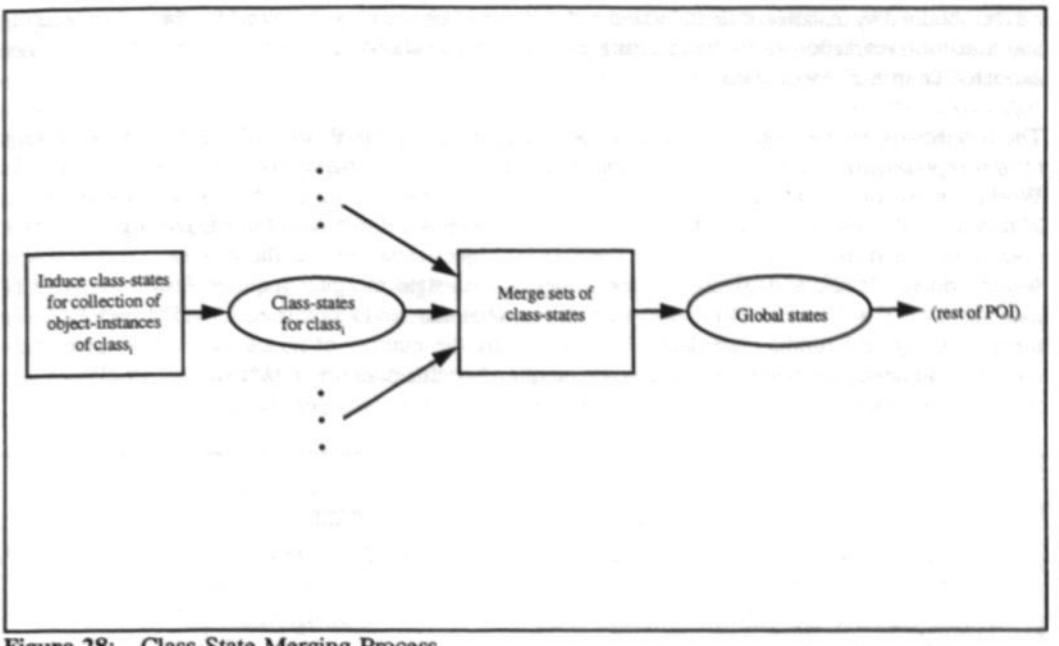

Figure 28: Class-State Merging Process.

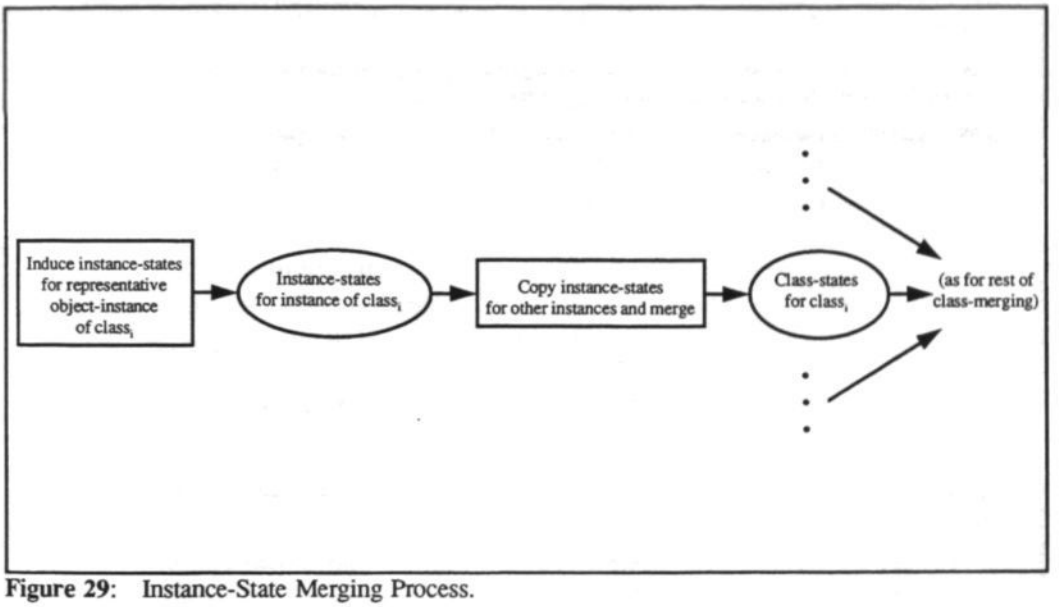

Figure 29: Instance-State Merging Process.

Partitioning and merging is a specialisation of the *divide-and-conquer* strategy. This strategy can be extended to a second stage. Class-states can themselves be obtained by inducing and merging *instance-states*, i.e., states for a single, representative instance of an object-class. In effect, the domain is partitioned by representative instances of each object-class. Class-states are obtained by inducing the instance-states for a representative instance, copying the set of instance-states for the representative object-instance to other instances of the same class (renaming the state-descriptions accordingly), and then merging the sets of instance-states. This process is known as *instance-state merging*. Global states

can be obtained by instance-state induction and instance-state merging, followed by class-state merging and transition extraction from the resulting global states, as shown in Figure 29. Transitions are then extracted from the global states.

The benefits of version-space merging can be illustrated using the Blocks World. Using the Nilsson (1980) representation, the domain description language for a one-hand, one-block, one-table Blocks World consists of nine RelationInstances. If unconstrained, the domain's version space would contain 2^9 nodes. Of the nine RelationInstances, two are owned by the `Hand` object-class, five by the `Block` object-class, and the remaining two by the `Table` object-class. Hence, the classes' version spaces would contain 2^2, 2^5, and 2^2 nodes, respectively. Class-state merging requires fewer nodes to be induced (i.e., $2^2 + 2^5 + 2^2 = 40$ nodes) than does global state induction (i.e., $2^9 = 512$ nodes). Since memory usage and runtime are determined largely by the number of nodes induced, version-space merging can be highly beneficial. Class-state merging benefits domains with many object-classes, and instance-state merging benefits domains with many object-instances per class.

System, technique, and author	Inputs	Outputs
(Not named); reinforcement learning; Doran (1968; 1969)	States	Desirabilities
STRIPS; macro-operator learning; Fikes, Hart and Nilsson (1972)	Triangle tables	Macro-operators
HACKER; repair learning; Sussman (1973)	Subplans	Generalised plan-bugs and repairs
LEX; induction (version spaces); Mitchell, Utgoff and Banerji (1982)	Positive and negative instances of states	Search control knowledge
MPS; macro-operator learning; Korf (1985)	Primitive moves and state-vectors	Macro-operators
CHEF; case-based planning; Hammond (1986)	Plans (cases)	Indexing
PET; experimental goal regression; Porter and Kibler (1986)	State-operator pairs	Heuristic rules
DYNA-Q; reinforcement learning; Sutton (1990)	State-action pairs	Q-values
PRODIGY; explanation-based learning (EBL); Carbonell and Gil (1990)	Failures, successes and interactions	Three types of rules
Rulemaker; sequence induction; Muggleton (1990)	State-action pairs	Decision tree of S-A pairs
GERRY; EBL and induction; Zweben, Davis, Daun, Draschler, Deale, and Eskey (1993)	Constraints (in scheduling problems)	Variable-ordering heuristics
Theo-Agent; EBL; Blythe and Mitchell, 1989	Planning problems and operators	S-A rules

Figure 30: Approaches to Learning Control Knowledge.

These benefits are not gained without paying a penalty. Additional computation is needed, in the form

of the merging sub-process. As merging also involves the construction of a lattice, the penalty can be significant. Where the number of object-classes in the domain, or the number of object-instances per class, or the number of RelationInstances owned by a class or instance is large, then combinatorial explosion is again encountered. Implementation has shown that it is vital to exploit the knowledge embodied in the domain constraints as early and as fully as possible in the merging process.

2.3.9 Other Approaches to Operator Learning

The combination of planning and learning capabilities is at the leading edge of research. The Foreword to a collection of key knowledge-based planning papers states:

> "... frontier problems [in planning] are suggested by the analogy between planning and learning. Learning occurs when an agent selects behaviour based on experiences in the world it inhabits. Planning occurs when an agent selects behaviour based on 'experiences' in an internal model of its world." ((Nilsson, 1990), p. xii of (Allen, Hendler and Tate, 1990)).

Approaches to learning in the planning context can categorised by the type of knowledge learned. Following Minton and Zweben (1993), I distinguish control knowledge from domain knowledge. Macro-operator techniques are on the borderline. Following Minton and Zweben, I regard macro-operators as control knowledge. Systems surveyed come from the knowledge-based planning and machine learning literatures. My survey confirms Minton and Zweben's statement (*ibid.*, p. 14) that most research to date has concentrated on learning control knowledge (see Figure 30).

It is instructive to sub-divide systems that learn domain knowledge (see Figure 31) into those that learn knowledge about the domain and those that learn knowledge about the actions available to the planner. The LIFE system uses mathematical induction and temporal generalisation to learn domain constraints, i.e., a form of knowledge about the domain. Like POI, LIFE takes state descriptions its input. However, LIFE does not use inductive techniques[31], nor does it output planning operators.

System, technique, and author	Inputs	Outputs
THOTH; generalisation; Vere (1978)	Before-and-after state-pairs, or state-sequences	Operators
Diffy-S; constructive induction; Kadie (1988)	Before-and-after state-pairs	Operators
LIFE; mathematical induction and temporal generalisation; Puget (1989)	States (dead-ends)	State-based domain constraints ("invariants")

Figure 31: Approaches to Learning Domain Knowledge.

Vere's (1978) THOTH and Kadie's (1988) Diffy-S systems are the only ones that learn planning operators. However, THOTH and Diffy-S differ from POI in that they require before-and-after state-

[31] Despite its name, mathematical induction is a deductive method (Puget, 1989).

pairs as input. Moreover, the sequencing information inherent in pairing a "before" state-description to its "after" state-description is essential for the algorithms in THOTH and Diffy-S. By contrast, POI is unique in not requiring such sequencing information in order to learn planning operators.

There is a third category of learning systems: those which learn plan-related knowledge (see Figure 32). All are designed to learn generic plans for later re-use. Moreover, all depend on the pre-existence of an operator-set, either as a generator of example plans or as a direct input to the learning system. None of these plan-learning systems is relevant to POI.

System, technique, and author	Inputs	Outputs
Structured rule induction; Dechter and Michie (1984a/b)	Plans	Generic plans
SteppingStone; EBL, induction and brute-force search; Ruby and Kibler (1989)	Goals and operators	Subgoal sequences, i.e., plans
BAGGER; schema-based planning and EBL; Shavlik (1989)	Operators and planning problem	Schemas, i.e., generic plans
(not named); EBL; Matwin and Morin (1989)	Plans	Plan-space constraints

Figure 32: Approaches to Learning Plans and Plan-Segments.

2.3.10 POI in Machine Learning Context

POI can be placed into the broader context of machine learning by classifying it in terms of Carbonell's (1989) design dimensions for inductive systems; see Figure 33.

Another aspect of POI, not considered elsewhere, is whether or not it employs *supervised learning* (Shavlik and Dietterich, 1990). Consideration shows that POI is both supervised and unsupervised, depending on the level at which the POI algorithm is viewed; see Figure 34. At the level of completeness and correctness of its training instances, the POI algorithm has an implicit supervisor[32], i.e., the real or simulated domain whose state is being observed. At the level of the state descriptions themselves, the POI algorithm is unsupervised. The observed state descriptions are not supplemented by information on the state-classes of which they are a member, let alone the planning operators - if any - which have been instantiated and executed in order to produce them. At the deeper level of each relationship expressed in a state description, the POI algorithm is again supervised in that naming conventions are exploited to identify relation names, object names, and the object-class to which each named object belongs.

[32] The supervisor may be a bad one in that it does not select examples for their teaching value.

Dimension	Classification of POI algorithm
Description languages	State descriptions for input, with states described as binary relationships between domain objects. Planning operators as output. Uses single-representation trick. Representational shift permitted.
Noise classification	Inputs assumed to be noise-free, i.e., complete, correct and positive.
Concept type	Domain constraints, in form of exclusion-rules, used as discriminant concepts. Concepts inherently characteristic because they are based only on positive instances.
Source of instances	Source is external to POI; may be either teacher or environment. POI algorithm limited to acquiring observations. No instance selection or experiment planning.
Incremental vs. one-shot	One-shot implementations, but could be modified to become incremental, as described.

Figure 33: Design Dimensions for POI Algorithm.

Level	Supervised?	Means of supervision
Training instance	Yes	Real (or simulated) domain can only express positive training instances. If Closed World Assumption is made, then instances that are not observed can be assumed to be negative.
State description	No	-
Relationship instance	Yes	Relationship and object names and their parent classes can be identified from naming conventions.

Figure 34: Levels of Supervision in POI Algorithm.

Normally, it is assumed in both supervised and unsupervised learning that a data-item cannot be a member of more than one class, i.e., the classes are mutually exclusive. This assumption is rarely made explicit. There are applications when the classes are clearly not mutually exclusive. For example, de Kleer and Williams (1987) were concerned with diagnosing multiple faults. The training instances would be the symptoms, and each training instance would be associated with a (potentially multiple) set of faults. POI assumes that training instances cannot be a member of more than one class.

2.4 LEARNING IN MULTI-AGENT SYSTEMS

This section reviews the Multi-Agent Systems literature on learning, emphasising the learning of planning knowledge in a multi-agent context. The literature is related to the research reported in this thesis.

2.4.1 Overview of Multi-Agent Systems

Multi-agent systems are a part of the field of Distributed AI. Bond and Gasser (1988, p. 3) define Distributed AI as "the subfield of AI concerned with concurrency in AI computations". They divide the world of Distributed AI into two primary arenas: Distributed Problem Solving and Multi-Agent Systems. Multi-Agent Systems research is concerned with coordinating intelligent behaviour between a collection of (possibly pre-existing) autonomous intelligent 'agents' which can coordinate their knowledge, goals, skills, and plans jointly to take action or to solve problems. Typical intelligent behaviours are to react appropriately to situations, to generate plans, and to learn.

Bond and Gasser (1988) do not provide a firm definition of an 'agent'. They state in a footnote (*ibid.*, p. 3, footnote 1) that they rely on a simple and intuitive notion of an agent as a computational process with a single locus of control and/or 'intention'. Searle (1980, footnote 2)[33] defines intentionality as:

> "that feature of certain mental states by which they are directed at or about objects and states of affairs in the world. Thus, beliefs, desires, and intentions are intentional states; undirected forms of anxiety and depression are not."

Bond and Gasser (1988) note that their view is very problematic, because agents may be implemented as concurrent processes, they may have multiple and conflicting goals, and the nature and reality of the concepts 'goal' and 'intention' are unclear. The process of defining the boundaries of what comprises an agent interacting with the world is also fraught with difficulties. For the purposes of this thesis, I define an *agent* as a software entity with autonomous processing capabilities and a private database, which acts on its environment on the basis of information it receives, perceives, retains and recalls. The agent's processing capabilities must include, as a minimum, the ability both to model its own current state and behaviour and to decide whether or not a new state its environment wishes to impose is acceptable. An agent with these minimal capabilities will be termed a *non-intentional* agent. Agents which, in addition, are capable of modelling the state and behaviour of other agents will be termed *intentional* agents. Agents with planning and/or learning capabilities are necessarily intentional.

The agents in a Multi-Agent System may be working towards a single global goal, or towards separate individual goals that interact. Goal-interaction may include conflict. Agents must share knowledge about problems and solutions. Crucially, "they must also reason about the *processes of coordination among the agents*" (Bond and Gasser, 1988, p. 3, italics in original). The task of coordination can be difficult, because there may be situations where there is no global control, globally-consistent knowledge, globally-shared goals, or global success criteria. There may even be no global representation of the system.

---

[33] Footnote 3 in the version reprinted in (Boden, 1990).

2.4.2 Learning as Timely Issue

Gasser and Huhns (1989b) list a number of timely research issues in Distributed AI. Beyond the early research by Huhns, Mukhopadhyay, Stephens and Bonnell (1987) into learning in document retrieval, Gasser and Huhns cite the approach of Shaw and Whinston (1989). In Gasser and Huhns' opinion, much more research into learning in Distributed AI is needed. In particular, they wish to reach the Distributed AI research goal first stated by Corkill (1982), Lesser and their colleagues: adaptive organisation self-design by a collection of problem-solvers or intelligent agents.

The research reported in this chapter goes part of the way to reaching this research goal. By closing the loop from planning operator induction through generative planning to plan execution, POIAgents are able to adapt to organisations of NonIntentionalAgents. They adapt in that, from knowing nothing whatsoever about the NonIntentionalAgents, POIAgents can learn enough by attempting to change the configuration of the NonIntentionalAgent organisation in order to induce planning operators, to generate a plan using the induced operators, and successfully to execute that plan to achieve a user-determined goal state. What the POIAgents cannot do is design new agent organisations, but this is outside the scope of my research. Interested readers should consult:

- Farhoodi, Profitt, Woodman and Tunnicliffe (1991) on using agents to model functional organisations.

- Doran (1990) on using Distributed AI to model the emergence of human organisations.

- Steels (1989) on using a self-organising, behaviour-based approach to tackle the problem of cooperation between distributed agents.

2.4.3 Knowledge-Richness of Learning

Shaw and Whinston (1989) describe the learning processes existing in Distributed AI systems, as well as presenting an architecture for incorporating learning capabilities in Distributed AI systems. They contrast learning in artificial neural networks from learning in Multi-Agent Systems. On the one hand, a neural network system learns by adjusting the weights of the interconnection links between nodes. On the other hand, a Multi-Agent System should learn by discovering new hypotheses and assessing existing hypotheses, eventually reaching a hypothesis strong enough to be the learned concept. In other words (*ibid.*, p. 415):

> "an agent in Distributed AI systems usually has a richer set of knowledge than the mere weights used in a neural-net node."

The multi-agent learning system described in this chapter exhibits a much richer set of knowledge than a neural network. This can be exemplified at several different levels:

- First, a POIAgent receives information from its multi-agent environment in the form of state descriptions. The POIAgent needs no prior knowledge whatsoever of the domain whose state is to be described, i.e., it can be naive. In a neural network, an input node would have to be provided for each possible state description. A domain state would be represented by switching these input nodes on and off. In sum, a neural network would be domain-specific.

- Second, a POIAgent extracts explicit domain knowledge from the input state descriptions in the form of lists of object-classes, object-instances, relationship-classes, and interrelationship

constraints. Moreover, the ontology for representing these lists - the entity-relationship model, supplemented by interrelationship constraints expressed as exclusion-rules - is also explicit. The fact that both the knowledge and its underlying ontology are explicit is evidenced by the facility for exchanging knowledge between agents. In the Message-Based Architecture testbed, the ontology is implemented in Smalltalk code in the form of the DomainClass, NonIntentionalAgent, Relationship, and ExclusionRule object-classes and the class hierarchy connecting them. In a neural network, equivalent knowledge could be extracted by hidden nodes, but both the extracted knowledge and the ontology on which it was based would remain implicit.

- Third, a POIAgent induces knowledge-based planning operators from the lists of object-classes, object-instances, relationship-classes, and interrelationship constraints. Once again, the induced planning operators and the STRIPS-based ontology for representing them are explicit. In the Message-Based Architecture testbed, the ontology is implemented in Smalltalk code in the form of the Activity object-class. A neural network would have to be provided with output nodes for each possible planning operator. The underlying ontology would again remain implicit.

2.4.4 Learning Processes

Shaw and Whinston (1989, p. 415) list the learning processes existing in a single agent as "the acquisition of new declarative knowledge, the development of problem-solving skills through instruction or practice, the organisation of knowledge into general, effective representations, and the discovery of new facts and theories through observation and experimentation." All of these processes are embodied in the Message-Based Architecture testbed, as follows:

- POIAgents acquire new knowledge in the declarative forms of lists of object-classes, object-instances, relationship-classes, interrelationship constraints, and planning operators.

- POIAgents develop their (domain-specific) problem-solving skills by observing or acting on the NonIntentionalAgents representing the problem domain. The testbed user instructs a POIAgent by choosing an appropriate series of goals to present to it.

- POIAgents with assimilation capabilities organise domain-specific knowledge of multiple agents into more general, effective representations.

- Naive POIAgents, or POIAgents with incomplete domain "knowledge", discover new facts and (domain) theories through observation and (user-guided) experimentation.

According to Shaw and Whinston (1989), there are two forms of learning activities: an agent can learn by improving its problem-solving skills (i.e., without contact with other agents), or it can learn by observing how other agents solve problems. Both forms of learning activities are implemented in the Message-Based Architecture testbed. Individual POIAgents can improve their problem-solving skills by means of the POI algorithm and learning-by-doing. Alternatively, POIAgents can learn by querying other agents and assimilating their responses, i.e., they can learn-by-being-told. The Message-Based Architecture testbed goes beyond Shaw and Whinston's prescription by enabling POIAgents to combine learning-by-doing with learning-by-being-told.

2.4.5 Single-Agent versus Multi-Agent Learning

Shaw and Whinston (1989) distinguish learning in Distributed AI systems (i.e., *multi-agent learning*) from the learning processes in a single agent (i.e., *single-agent learning*). They identify two forms of multi-agent learning: emerging intelligence and group discovery. With the assimilation capabilities, groups of POIAgents become capable of achieving more tasks than the sum of the tasks which can be individually achieved by the POIAgents. One subgroup of POIAgents may specialise in collecting evidence, another in generating hypotheses, and a third evaluating hypotheses in order to discover new concepts collectively. When embedded in an agent, Part 1 of the POI algorithm implements the collection of evidence and the generation of hypotheses in the form of lists of object-classes, relationship-classes, and interrelationship constraints. Part 2 of the POI algorithm continues hypothesis generation to obtain knowledge-based planning operators. The assimilation process includes the evaluation of other agents' hypotheses and the generation of hypotheses based on jointly-collected evidence.

I part company with Shaw and Whinston (1989) in respect of the approach to multi-agent learning. Their claim is that Distributed AI systems should adapt and evolve like natural systems do, by letting agents compete with one another. Learning is achieved genetically, i.e., by reproducing only those agents with a high fitness score. Their agents do not have single-agent learning capabilities. By contrast, my approach is to achieve multi-agent learning by providing agents with both single-agent learning capabilities (in the form of the POI algorithm) and knowledge-sharing and assimilation capabilities.

2.4.6 Role of Cooperation

As in this thesis, Sian (1991) takes the approach that agents are persistent objects which actively learn from their experiences. In investigating the problem of dynamic adaptation in multi-agent systems, he observes that effective learning requires cooperation between agents. His paper addresses the motivation for cooperation, the effects of the problem structure, the levels at which cooperation can take place, the means by which learning agents can cooperate, the learning strategies employed, and the issues involved in using a consensus model for learning. The aspect that is most relevant to this thesis is his identification of the possible levels at which cooperation may take place.

Sian (1991) depicts the levels within an agent as shown in Figure 35. Since modifications in values and beliefs tend to be local decisions within an agent, cooperation for problem-solving is more likely to occur at the levels of knowledge, actions, planning, and goals. Attributes necessitated by a multi-agent environment, such as organisation and communication, are also amenable to learning. He notes that, the higher up the hierarchy that one considers cooperative learning, the greater is the sophistication of what is being learned and the lesser is the volume of the communication traffic.

Only the knowledge and actions levels are relevant to this thesis. Sian (1991) splits the knowledge level into two sub-levels. The lower sub-level of knowledge exchange would be an exchange of facts. In the case of the Message-Based Architecture testbed, this would involve the exchange of the raw observations made by the POIAgents. I have chosen not to implement knowledge exchange at this level, largely because of the volume of information that would have to be stored and communicated.

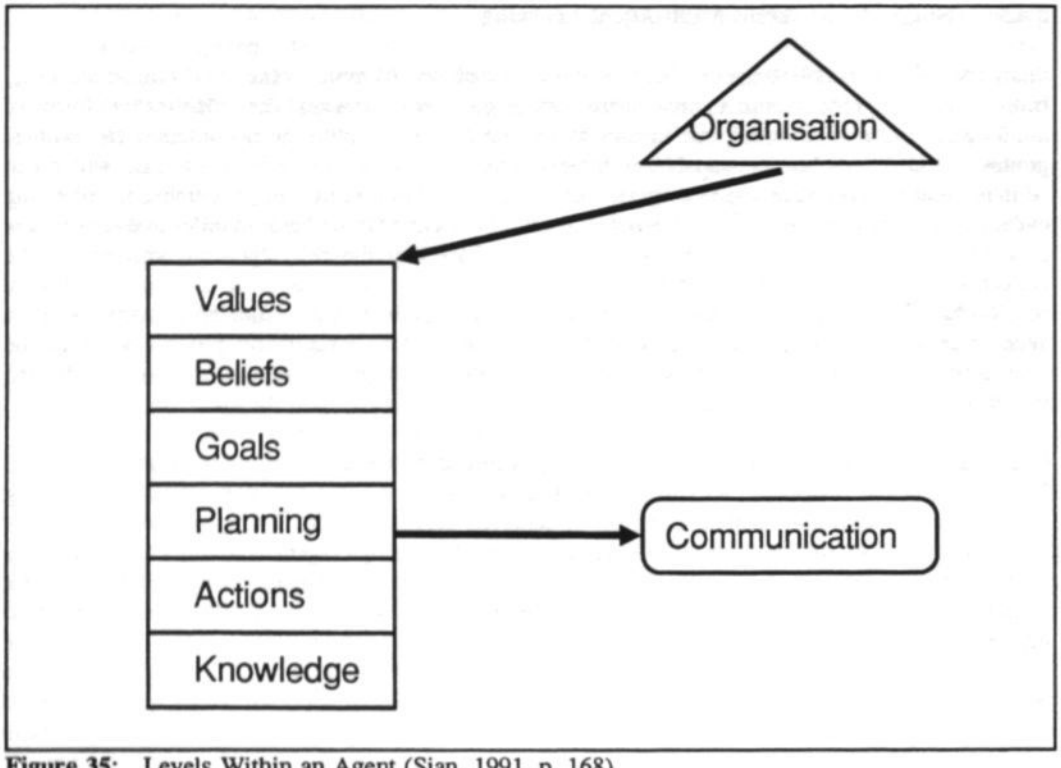

Figure 35: Levels Within an Agent (Sian, 1991, p. 168).

Sian (1991) notes that, at the higher sub-level, agents may cooperate in the learning of rules based on the lower-level facts[34]. In the Message-Based Architecture testbed, this level of cooperation is implemented as the exchange of interrelationship constraints in the form of exclusion-rules. My research has shown that it is necessary to supplement the exchange of rules by also exchanging the associated domain model, e.g., the object- and relationship-classes.

At the actions level, Sian (1991) notes that agents may cooperatively try to find generalised descriptions of actions that result in the successful execution of plans. In the Message-Based Architecture testbed, actions are generalised as knowledge-based planning operators. My research has shown how this can be done using the POI algorithm. Exchange of the planning operators between agents is outside the scope of my research, but could be readily implemented.

2.4.7 Learning Planning Knowledge

In the past three or four years, a number of authors have addressed the learning of planning-related data-structures in a multi-agent context. As in the single-agent context (see Chapter 2), all the algorithms published to date require prior domain knowledge in the form either of primitive operators or of previously-generated plans. They acquire control knowledge, rather than domain knowledge. For example:

---

[34] His paper states " ... inductive learning ...", but notes elsewhere that other learning strategies are possible.

- The ARMS system (for *Acquiring Robotics Manufacturing Schemata*) relies on using a domain theory or a priori domain knowledge to analyse a previous solution to discover why or not the solution was successful (Segre, 1991). Explanation-Based Learning (EBL) techniques are used in learning how to assemble new mechanisms by observing and analysing another agent's solution to sample assembly problems. The outputs of EBL are macro-operators.

- Robo-Soar (Laird, Yager, Hucka and Tuck, 1991) is a system implemented in the general symbolic Soar architecture that performs simple block manipulation and button-pushing tasks using a Puma robot arm and a camera-based vision system. The tasks are quite simple, but serve as a testbed for investigating issues in external interaction. The robot's goal is to line up a set of small blocks that are scattered over the work area while monitoring a light. If the light goes on, a button must be pushed as soon as possible to turn the light off. Robo-Soar "starts with knowledge of the individual operators for the robot arm, such as opening and closing the gripper, but it has no control knowledge for deciding when the operators are appropriate ... It learns this knowledge from experience and outside guidance" (Laird, Yager, Hucka and Tuck, 1991, p. 114).

- PRODIGY (Minton, Carbonell, Etzioni, Knoblock and Kuokka, 1987) is a flexible planning system that encodes its domain knowledge as declarative operators. It applies the operator refinement method to acquire additional preconditions or postconditions when observed consequences diverge from internal expectations (Carbonell and Gil, 1990). When multiple explanations are obtained, experiments are generated to discriminate between them. Carbonell and Gil differentiate four types of experimentation by the information acquired. Experiments may be formulated to augment an incomplete domain theory, to refine an incorrect domain theory, to acquire search control knowledge in an otherwise intractable domain theory, or to acquire or correct factual knowledge about the external state of the world. Using PRODIGY, they investigate a series of methods for learning by experimentation in the context of planning to acquire search control knowledge and to acquire factual knowledge.

2.4.8 Knowledge Refinement versus Theory Formation

Carbonell and Gil (1990, p. 193) believe that "automated knowledge refinement is a very important aspect of autonomous learning, and one where success is potentially much closer at hand than the far more difficult and seldomly encountered phenomenon of formulating radically new theories from ground zero." As I claim that planning operators can be induced *ab initio*, then my research is specifically concerned with theory formation.

2.5 CHAPTER 2 SUMMARY

This chapter has reviewed the relevant literature in the fields of software engineering, knowledge-based planning, machine learning, and multi-agent systems.

The POI ontology is based on representations from software engineering, supplemented (where necessary) by knowledge-based planning representations. The types of knowledge that need to be represented in knowledge-based plan generation are identified by reference to Tate, Hendler and Drummond's (1990) definition of plan generation. Domain structure is modelled using Chen's (1976) entity-relationship model, enhanced by means of Nijssen and Halpin's (1989) exclusion constraints. Domain behaviour is modelled using state-transition networks, as found in software engineering.

I show that representations similar to POI's structural and behavioural models are to be found in the knowledge-based planning literature. The features of the POI ontology which are unusual from the planning viewpoint are:

- the description of states in terms of inter-object relations,
- the representation of plans as sequences of states, rather than sequences of transitions, and
- the emphasis on representing domain constraints.

The POI algorithm encapsulates an inductive technique from the machine learning field known as the *version space and candidate elimination* algorithm (Mitchell, 1982). The inductive learning performed in the POI algorithm has been related to Simon and Lea's (1974) *two-space model*. Instance selection and experiment planning fall outside the scope of POI. The *single-representation trick* (Cohen and Feigenbaum, 1982) is employed. Justification is given for modifying the candidate elimination algorithm for application in POI. POI's induction step is described in its incremental form, although a one-shot version is implemented.

Other learning systems in the planning area are surveyed, showing that they either learn control knowledge or make use of sequencing information (e.g., before-and-after state-pairs, state-sequences, or previous-generated plans). The POI algorithm is unique in learning planning operators without needing sequencing information. It formulates domain theories from scratch.

Finally, multi-agent systems techniques are used in closed-loop testing of the POI algorithm. Learning is a timely issue in distributed AI, of which multi-agent systems are a part. The knowledge-richness of learning is discussed. Learning processes are listed, and single- and multi-agent learning are contrasted. The role of inter-agent cooperation is summarised. Other multi-agent systems that learn planning knowledge are shown to acquire control knowledge, rather than domain knowledge.

3 PLANNING OPERATOR INDUCTION (POI)

The purpose of this chapter is to document the Planning Operator Induction (POI) algorithm and its underlying ontology.

In Chapters 1 and 2 the purpose of, and requirements for, the algorithm have been discussed. Section 3.1 draws these requirements together. Section 3.2 presents the POI ontology, which links structural and behavioural domain models by means of a linking model. Section 3.3 describes the functional decomposition of the POI algorithm. Section 3.4 briefly describes how the ontology and algorithm have been implemented. Section 3.5 lists the known limitations, of which combinatorial explosion is the most important. Section 3.6 describes the countermeasures adopted to combat the combinatorial explosion. Finally, Section 3.7 summarises the chapter, comparing the POI algorithm with related research.

This chapter describes the full, nine-step POI algorithm. The full algorithm is needed when the input takes the form of a set of world-state descriptions, as used in the closed-loop experiments described in Chapter 5. In Chapter 4, open-loop experiments are described in which the first three steps (i.e., Part (1) of the POI algorithm) are modified. The modifications are described in Chapter 4. Both full and modified forms of the POI algorithm employ the POI ontology which is described in this chapter.

3.1 REQUIREMENTS

There are three groups of requirements on the POI ontology and algorithm:

(1) Requirements essential to provide the necessary functionality.

(2) Requirements needed to ensure potential real-world application.

(3) Requirements introduced to make the research tractable.

3.1.1 Functionality Requirements

The following set of functionality requirements has been extracted from the statements in Chapter 1:

R1: The input to the Induction part of the POI algorithm shall take the form of a list of domain objects (and their classes), a list of inter-object relationship-classes, and a list of interrelationship constraints.

R2: Optionally, the lists mentioned in R1 shall be obtained (in the Acquisition part of the POI algorithm) by converting a set of descriptions of domain states.

R3: The POI algorithm shall output a set of planning operators. In this thesis, the planning operators are represented using the STRIPS planning-operator formalism.

R4: The POI algorithm shall employ induction techniques based on the version space and candidate elimination algorithm (Mitchell, 1982).

R5: The POI algorithm shall not be dependent on the existence of information resulting from previous plan generation or execution processes for the same domain or for an analogous domain. In particular:

R5a: The POI algorithm shall not be dependent on the input state descriptions being adjacent, i.e., sharing transitions in common. Adjacent state descriptions would amount to plan segments or subplans.

R5b: The POI algorithm shall not be dependent on any ordering of the input set of state-descriptions. The ordering might be interpreted - correctly or incorrectly - as dependencies between states.

3.1.2 Application Requirements

Additional requirements were needed to ensure that my research has potential application to real-world domains:

R6: It shall be possible to apply the POI algorithm to complex, real-world domains using Commercial-Off-The-Shelf hardware, operating systems, and computer languages.

R7: Techniques developed to ensure the real-world applicability of the POI algorithm (e.g., measures for countering the combinatorial explosion) shall be domain-independent, but use domain-specific knowledge.

R8: The domain knowledge used in the countermeasures shall be meaningful to users of the POI *algorithm. In particular,* the countermeasures must not be artefacts of the domain representation.

The HPCE domain is introduced in Chapter 4 to verify that **R6** to **R8** have been satisfied.

3.1.3 Tractability Requirements

Further requirements were introduced in order to make the research tractable. Such requirements introduce assumptions or limitations which future researchers may wish to relax. The assumptions and limitations are as follows:

R9: The POI algorithm shall be based on the assumption that the input information supplied to it is complete. This assumption allows the POI algorithm to exploit the Closed World Assumption (Reiter, 1978).

R10: The POI algorithm shall be based on the assumption that the input information is consistent, correct, and precise. This assumption allows the POI algorithm to dispense with computing probabilities, fuzzy sets, Certainty Factors, and the like.

R11: The POI algorithm shall be limited to processing descriptions of discrete objects, discrete relationships, and discrete constraints. In other words, the POI algorithm shall not process metric information.

R12: By default, the POI algorithm shall assume that each change in world state (i.e., a world

transition) arises from the action of a single initiating agent: the *actor*. This assumption supports the Single Actor part of the Single-Actor/Single-State-Change (SA/SSC) meta-heuristic. N.B. This default may be relaxed if no planning operators are induced.

R13: The POI algorithm shall not assume that the actor is the same in all transitions.

R14: By default, the POI algorithm shall assume that the actor in a transition changes its state in respect of only one relationship. This assumption supports the Single-State-Change part of the SA/SSC meta-heuristic. N.B. This default may be relaxed if no planning operators are induced.

R15: The POI algorithm shall not assume that an agent's state-changes associated with those transitions in which it is the actor all relate to the same relationship-class.

R16: The POI algorithm shall assume that changes in the state of the world occur as a linear sequence of transitions. In other words, concurrent action is excluded.

R17: The POI algorithm shall assume that a domain consists of a finite number of object-classes, each with a finite number of instances.

R18: The POI algorithm shall represent inter-object relationships as binary relationships.

R19: The POI algorithm shall assume that there are a finite number of relationships between each pair of object-classes.

R20: The POI algorithm shall represent interrelationship constraints as binary constraints.

Future researchers may also regard requirements other than **R9** to **R20**, notably **R4**, as assumptions or limitations to be relaxed. However, the complete set of requirements, as listed above, are the baseline for this thesis.

3.2 POI ONTOLOGY

The POI ontology is based on connecting a *structural* model of a domain (or universe of discourse) to its *behavioural* model by means of a *linking* model. All three models are declarative, although the behavioural model has a procedural reading. All sets are finite.

Figure 36 shows the complete POI ontology using Chen's (1976) entity-relationship diagram notation. The structural, behavioural, and linking models are shown as partitions of the complete ontology. Each model is described in a separate sub-section below.

Note that the POI ontology is both:

- presented as an entity-relationship model, and
- capable of taking its input in the form of an entity-relationship model.

Hence, the POI ontology is a *meta-representation*, i.e., a representation that is capable of representing itself.

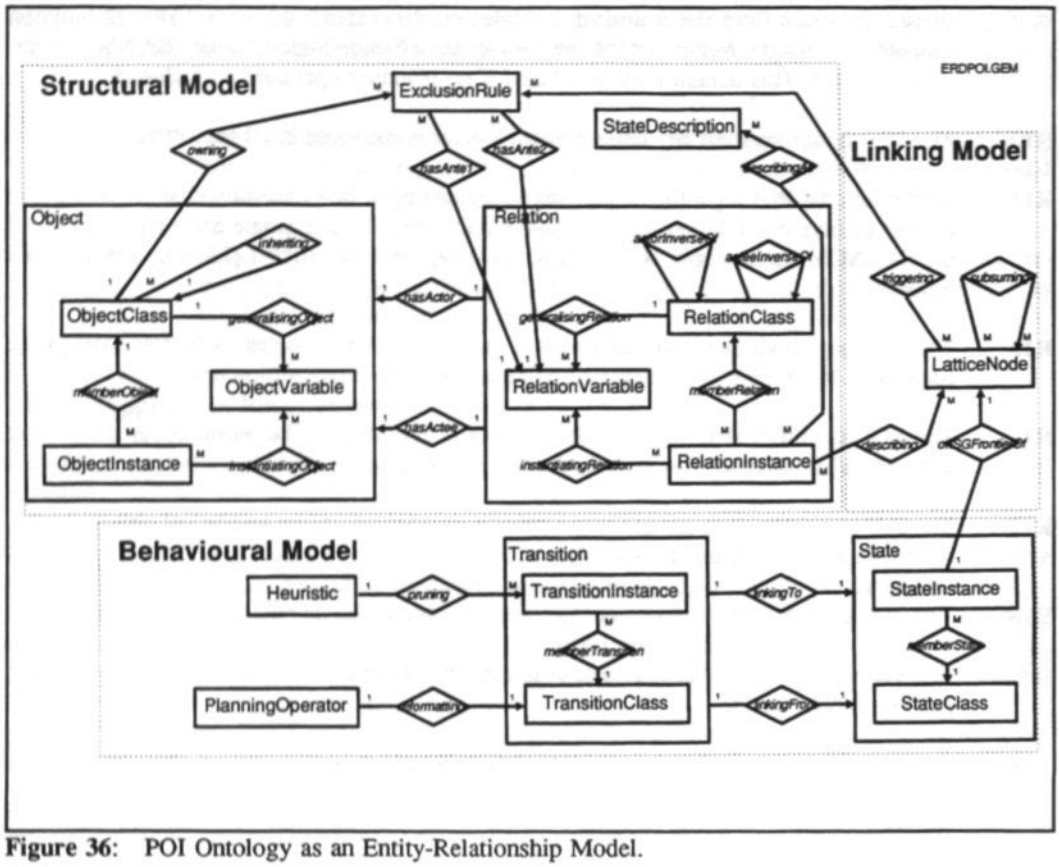

Figure 36: POI Ontology as an Entity-Relationship Model.

3.2.1 Structural Model

The structural model has four components:

- a set of *objects*, which model the domain objects.
- a set of inter-object *relationships*, which model the real-world relationships between domain objects.
- a set of interrelationship *constraints*, which model the domain constraints.
- a set of *world-state descriptions*, from which objects, relationships, and constraints may be extracted if they are not provided directly.

Objects and relationships have unique names. There is no requirement at the ontological level for constraints and world-state descriptions to be named.

The structural model is an extension of Chen's (1976) entity-relationship model, in that:

- entities in Chen's model are restricted to representing domain objects.

- a representation for domain constraints has been added.

Unlike a number of other extensions (e.g., NIAM (Nijssen and Halpin, 1989) and Ontolingua (Gruber, 1992)), the structural model does not permit relationships to be treated as objects. Consequently, there are no second-order relationships. Moreover, the objects do not have attributes (or properties). The only attribute of relationships is a truth value. If necessary, domain attributes may be represented as additional objects, together with appropriate relationships. For example, suppose one wished to refer to a "red car". Then an object named `red` would be defined, and the `car` object related to it by a relationship named (say) `isColoured`.

The set of objects is partitioned, with each partition representing an object-class. An object-class is also a type. Similarly, the set of relationships is partitioned into relationship-classes. Object-classes and relationship-classes are equivalent to Chen's (1976) entity-sets and relationship-sets, respectively. There is a predicate associated with each object-class to determine whether an object belongs to it. Similarly, a predicate is associated with each relationship-class to determine whether a relationship belongs to it.

Relationships associate pairs of objects, i.e., relationships are strictly binary. Relationships are directed, i.e., asymmetric; the object-pairs are ordered. The first object in the pair is said to be the relationship's *actor*, and the second object is its *actee*. An object is said to be *mentioned* in a relationship if it is either the actor, or the actee, or both. Relationship-classes are typed by the classes of the first and second object in the pair. There is a constraint between each relationship-class and its instances: the actor and the actee must belong to the corresponding object-classes associated with the parent relationship-class. Every primary relationship has associated with it a *converse*, e.g., the relationship-class named `holding` would have an associated converse `heldBy`. The converse has the same truth value as its primary. Other than asymmetry, no other constraints are placed on relationships at the ontological level. Both the primary relationship-class and its converse have inverses, which have the opposite truth value to their primary/converse. The inverse of `holding` would be `notHolding`, and the inverse of `heldBy` would be `notHeld`.

Object-variables are subsets of objects all belonging to the same object-class. Similarly, *relationship-variables* are subsets of relationships all belonging to the same relationship-class. In a relationship-variable, either the first object, or the second object, or both, are object-variables. A relationship-variable in which both objects are object-variables is known as a *clause*.

The set of *constraints* contains all feasible pairs of clauses. To be feasible, the clauses in a pair must be different, but with at least one object-variable in common. The object-variable in common is known as the *pivot* object-variable. The object-class(es) of the pivot object-variable is said to be the *owner(s)* of the constraint. Since constraints are not directed, the clause-pairs are unordered. A constraint is said to *match* a set of relationships if the constraint's clause-pair contains a pair of relationships, both of which are included in the set of relationships being matched.

World-state descriptions are subsets of the set of relationships. Any constraint which matches any world-state description is said to be UNUSEABLE.

A digression concerning negation of relationships is needed here. During my research I weighed the merits of treating the absence of a primary relationship at a meta-level, as do Ramsay and Barratt (1987). The advantage would be that mutual exclusion-constraints would not be needed. However, I chose to treat the negation of relationships by defining inverse relationships, for the following reasons:

- States can be more concisely described in domains with multiple object-classes and -

instances. For example, describing a blocks world state in which all hands are empty requires one (inverse) relationship-instance per hand and one (inverse) relationship-instance per block. By contrast, negation at the meta-level would require a number of relationship-instances equal to the product of the number of hands and the number of blocks.

- Determination of the absence of a relationship-instance is more efficient. Instead of checking that all possible instances of a given relationship-class have been negated, it is sufficient to check for the truth of a single inverse relationship-instance.

3.2.2 Behavioural Model

The behavioural model has four components:

- a set of *states*.
- a set of *transitions*.
- a *meta-heuristic*.
- a set of *planning operators*.

States are subsets of the set of relationships such that:

- all the objects in the domain are mentioned at least once.
- any constraint which matches any relationship in the subset is UNUSEABLE.

State-instances are generalised into state-classes by replacing relationship-instances by relationship-classes.

The meta-heuristic determines which pairs of states are to be associated with transitions. Transition-instances are generalised into transition-classes by replacing state-instances by state-classes.

The states and transitions form the state-transition network which models the behaviour of the domain. Plans can be extracted as directed paths through this state-transition network, starting at the initial state and ending at the goal state. It is possible to have multiple paths (i.e., alternative plans) between a given pair of states.

Planning operators are reformatted transition-classes. The preconditions list of a planning operator is the intersection of the relationship-variables describing the transition-class's state-class-pair. The add-list is the set-subtraction of the relationship-variables describing the transition-class's `linkingTo` state-class from the relationship-variables describing its `linkingFrom` state-class. The delete-list is the set-subtraction of the relationship-variables describing the transition-class's `linkingFrom` state-class from the relationship-variables describing its `linkingTo` state-class. The planning operator's variable-list is the union of all object-variables mentioned in its preconditions-, add- and delete-lists.

3.2.3 Linking Model

The linking model has one component: a finite lattice of *nodes* in the domain's rule space. This lattice will be termed the domain's *version-space lattice*[35]. The version-space lattice is a knowledge structure consisting of a set of descriptions, a partial ordering, and *greatest lower bound* and *least upper bound* operators. The partial order must express a *subsumption* relationship. One node subsumes another if the descriptions of the latter node is a proper subset of the descriptions of the former node, i.e., the former node *specialises* the latter node.

In the POI algorithm:

- the descriptions are relationship-instances,
- the version-space lattice has the node described by zero relationships as its bottom element, i.e., the most general element.
- the greatest lower bound operator just returns the bottom element.
- any node described by a set of relationship-instances which triggers one or more constraints is said to be *over-specialised.*
- any node that would be subsumed only by over-specialised nodes is said to be a *top element.* The set of top elements is the *SG frontier.*
- nodes that are not top elements but which have two or more top elements are said to be *ambiguous.*
- each top element is associated with a *possible state.* To be accepted as a state-instance, a possible state must describe all known object-instances completely.

The linking model is connected to the structural model through the relationship-instances, used to describe the lattice node, and through the constraints, used to prune those sets of descriptions that would lead to a node becoming over-specialised. The linking model is connected to the behavioural model through the set of state-instances associated with the top elements. The set of state-instances is the domain's *state-space.*

For planning purposes, only version-space lattices which have multiple top elements are interesting. There can be no transitions (and hence no planning operators) if there is just one top element, i.e., the state-space contains just one state-instance.

---

[35] If the complete lattice were to be built, it should be called the rule-space lattice. However, only the version space is built in POI.

3.3 ALGORITHM

This section specifies the processing performed by the nine-step POI algorithm using the ontology described in the previous section. First, the overall processing strategy is described. This strategy is diagrammed as a state-transition network, with steps in the POI algorithm being mapped onto the states and transitions. Second, each step in the baseline version of the POI algorithm is described in more detail. Where necessary, a step will be further decomposed, on the basis of the description of the overall processing strategy.

3.3.1 Processing Strategy

The processing strategy in the POI algorithm is to construct the structural model, then the linking model, and finally the behavioural model. Requirements **R1**, **R2** and **R3** determine that the POI algorithm shall be decomposed into two functional parts (see Figure 37). The first part constructs the structural model by converting observed world-states into the three-element list of domain objects, relationships and constraints. The second part constructs the linking and behavioural models by inducing a set of planning operators from such a three-element list, using the meta-heuristic for identifying valid transitions. The single-agent POI application described in Chapter 4 needs only the second part of the algorithm, but the multi-agent POI application in Chapter 5 uses both parts of the algorithm.

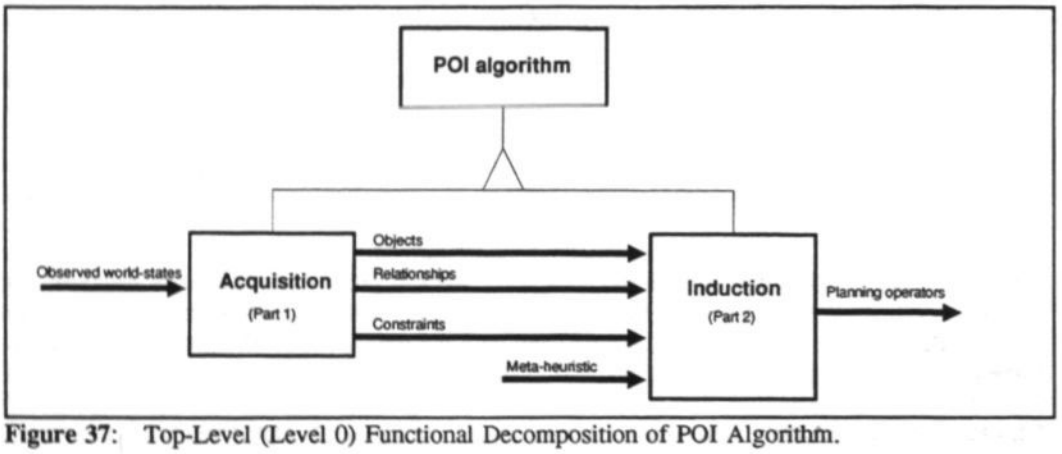

Figure 37: Top-Level (Level 0) Functional Decomposition of POI Algorithm.

Each model is constructed by visiting its entity-classes in turn, instantiating all the valid entity-instances for that entity-class. Valid entity-instances are instances which satisfy all the relationships and their cardinality and exclusion constraints. The order in which the entity-classes are visited is constrained by the relationship-classes; the next entity-class to be visited must be adjacent to the entity-class which has just been instantiated. *Adjacent* entity-classes are pairs of entity-classes which are linked directly by one or more common relationships. For example, RelationClass and RelationVariable are adjacent, but RelationClass and ExclusionRule are not. Having instantiated an entity-class, it is not necessary to return to it. Since there is a finite number of entity-classes (namely 14 of them), each with a finite number of instances, the POI algorithm is guaranteed to terminate.

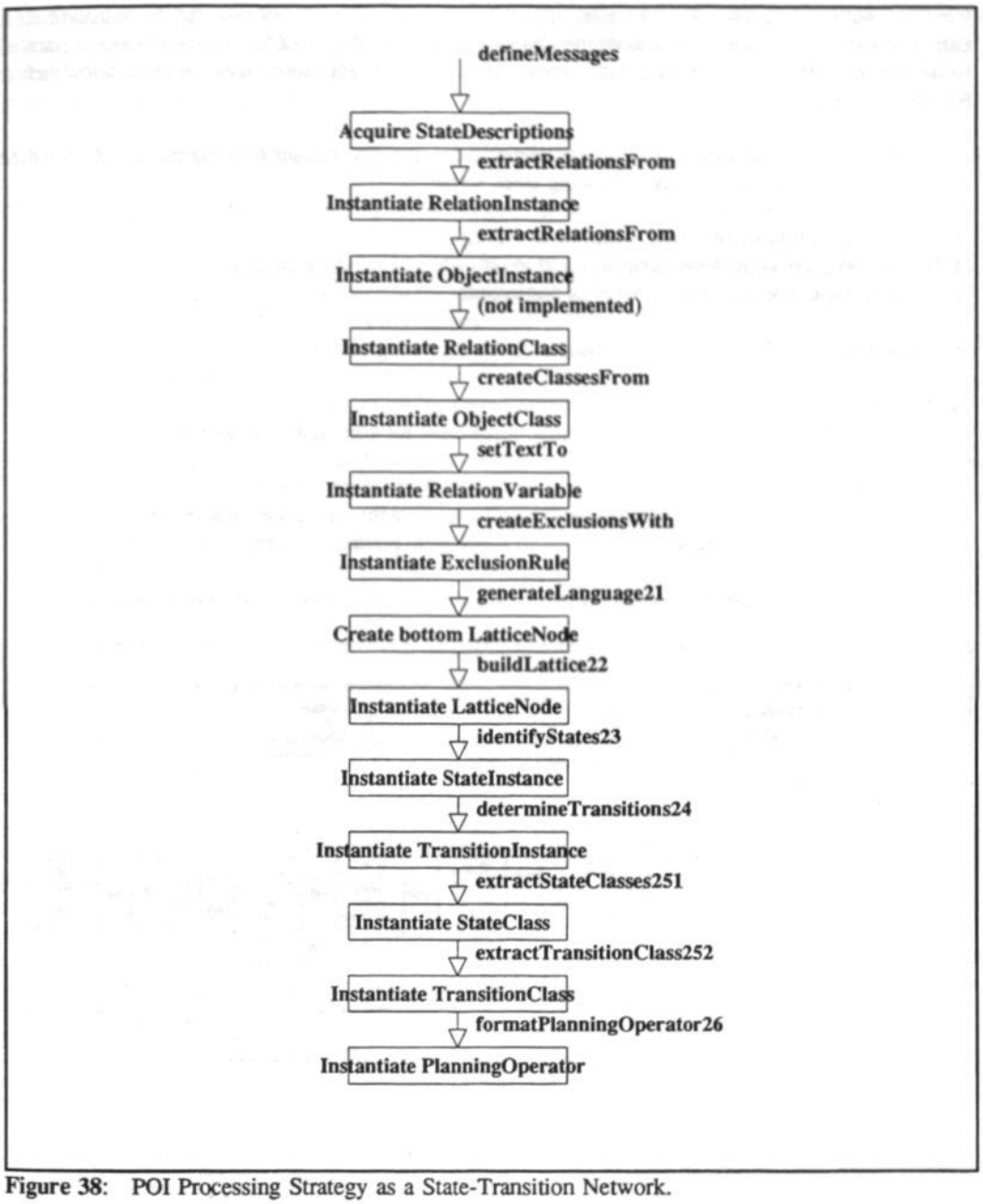

Figure 38: POI Processing Strategy as a State-Transition Network.

The POI algorithm may be represented as a state-transition network, as shown in Figure 38, with states as nodes and transitions as arrows. The mapping to steps in the POI algorithm is shown on the left-hand side, with the corresponding names of the implemented Smalltalk methods shown as transitions on the right-hand side. Each state in the state-transition network corresponds to the process of instantiating an entity-class, and each transition corresponds to the transfer of the focus of attention to an adjacent entity-class. The network is acyclic because each entity-class is visited only once. Parallel instantiation of some entity-classes is possible, although this cannot be shown using the state-

transition network notation. For example, once the RelationInstance entity-class has been instantiated, then it would be possible to instantiate the ObjectInstance and RelationClass entity-classes in parallel (or in either order). Figure 38 shows the linearisation in which ObjectInstances are instantiated before RelationClasses.

Each part of the POI algorithm may be functionally decomposed further (see Figure 39). Acquisition (Part 1) can decomposed into the following three steps:

(1.1) Acquisition of observed world-state descriptions.
(1.2) Recognition of domain objects and relationships, including classes.
(1.3) Compilation of interrelationship constraints.

and Induction (Part 2) can be decomposed into the following six steps:

(2.1) Generation of world state-description language.
(2.2) Construction of version space using constraints for candidate elimination.
(2.3) Identification of world states from version-space nodes.
(2.4) Determination of valid transitions using one of the meta-heuristics.
(2.5) Generalisation of world states and transitions to state- and transition-classes.
(2.6) Formatting of transition-classes as STRIPS-style planning operators.

Several of these steps are decomposed further in the course of detailing the POI algorithm.

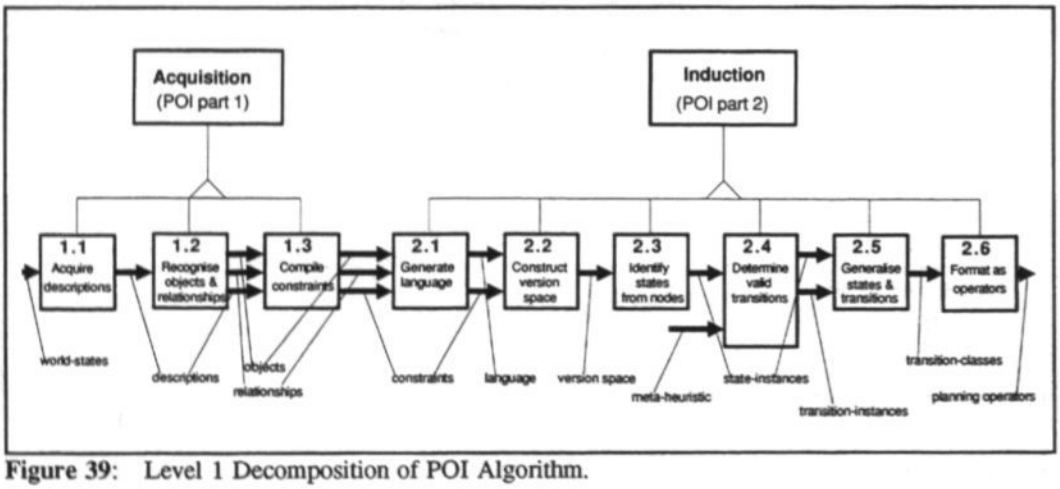

Figure 39: Level 1 Decomposition of POI Algorithm.

3.3.2 Step (1.1): Acquire Descriptions

In Step (1.1), the description of each observed world state is acquired and expressed in terms of StateDescription instances. On completion of Step (1.1), the POI algorithm will have acquired a set of world-state descriptions expressed in terms of an unknown domain-dependent description language. The language will be implicit in the StateDescription instances. Since - by assumption - each observed world-state is complete and the observed set of descriptions is consistent, the implicit description language will be consistent and complete (by comparison with the observed world-states).

The processing done in Step (1.1) is straightforward. A new StateDescription instance is created for

each observed world state. Its *descriptions* attribute is assigned a value whose data-type is a set of one or more elements, where each element is itself an ordered list of two or three words. The value of the descriptions attribute represents the observed world state is such a way that the following domain-independent meta-knowledge holds true:

(1) The state of the domain is represented in terms of binary relationships.

(2) Each relationship is described as an ordered list of two or three words.

(3) A prefix syntax is used to describe relationships. Hence, the first word in each relationship description represents the name of the relationship, e.g., `pressing`, `holding`, `powering`, etc. The relationship-name is mandatory in all relationship descriptions.

(4) The relationship is directed, such that the second word names the domain object which causes or performs the relationship, e.g., `finger1`, `hand3`, `powerSupply2`, etc. This domain object is termed the relationship's *actor*. The actor-name is mandatory in all relationship descriptions.

(5) If present in the relationship description, the third word names the domain object upon which the relationship is performed, e.g., `key88`, `block2`, `capillary6`, etc. The domain object named by the third word is termed the relationship's *actee*. The presence of an actee indicates that the relationship is a primary or a converse relationship.

(6) If the relationship description has no actee, then the relationship named by the first word is an inverse relationship. The name of the corresponding primary or converse relationship may be obtained by removing the first three letters - which must be the string "not" - from the first word and converting the (new) first letter to lower case. For example, the primary relationship identified from the relationship description `notPressing finger3` would be named `pressing`.

The POI algorithm is not dependent on the precise nature of the representation of relationship description defined above, providing that it is possible to identify from the relationship description:

- Whether the relationship is primary or inverse,
- The name of a type of primary relationship directed from actor to actee,
- The name of the actor object, and
- The name of the actee object (if the relationship is primary).

For example, infix or postfix syntax could be used. Alternatively, a form of structured natural language could be used, e.g., "The `power supply 2` is `powering` the `capillary 6`.".

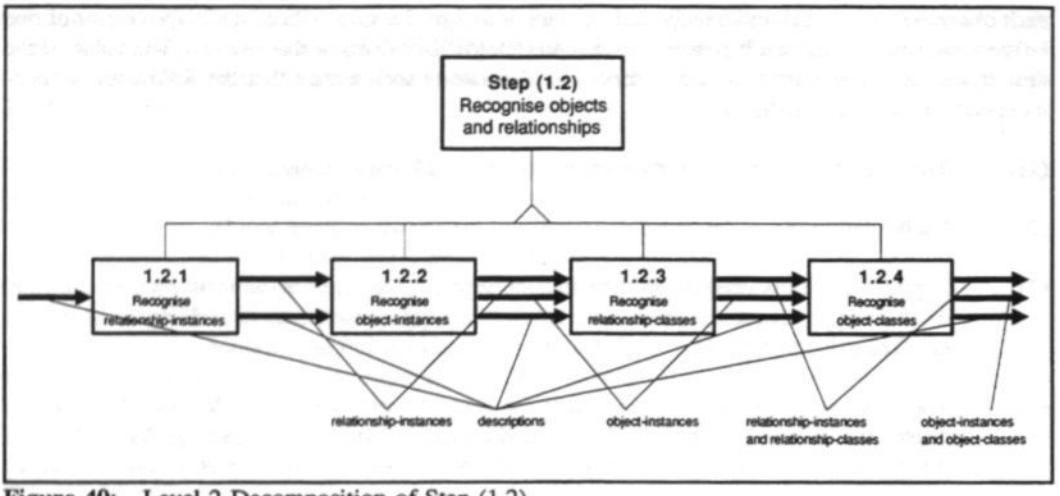

Figure 40: Level 2 Decomposition of Step (1.2).

3.3.3 Step (1.2): Recognise Objects and Relationships

In Step (1.2), the acquired StateDescription instances are processed to extract the instances of the ObjectClass, ObjectInstance, RelationInstance, and RelationClass entity-classes. In the POI algorithm this is done by exploiting the domain-independent meta-knowledge described in the previous subsection. After Step (1.2), the POI algorithm has acquired domain knowledge equivalent to an application-level entity-relationship model, but without the 1-to-1, 1-to-n, m-to-1, or n-to-m mappings.

As shown in Figure 40, the processing done in Step (1.2) can be decomposed further. In Step (1.2.1), each element of the descriptions attribute of each newly-created StateDescription instance is compared with the already-existing RelationInstance instances. If an element cannot be matched to any existing RelationInstance instance, then a new RelationInstance instance is created. In Step (1.2.2), the second and (if present) third words of each description are compared with the already-existing ObjectInstance instances. If any such word cannot be matched to the name of an existing ObjectInstance instance, then a new ObjectInstance instance is created. Cross-references are made between the identified or newly-created ObjectInstance instance and any RelationInstance instance in which it takes part.

In Step (1.2.3), the newly-created RelationInstance instances are compared with already-existing RelationClass instances. If any RelationInstance instance cannot be matched to an existing RelationClass instance, then a new RelationClass instance is created. In Step (1.2.4), the inverse and converse RelationClass instances are identified or, if necessary, created. Cross-references are made between the RelationClass instance and its corresponding RelationInstance instance(s). In Step (1.2.5), a procedure similar to Step (1.2.3) is followed to identify and cross-refer the ObjectClass instances corresponding to newly-created ObjectInstance instances. The ObjectClass and RelationClass instances are also cross-referred.

3.3.4 Step (1.3): Compile Constraints

In Step (1.3) the domain constraints (i.e., ExclusionRule instances) are compiled from the RelationClass instances extracted in Step (1.2). In the POI algorithm, this is done by:

- Forming all pairs of RelationClass instances which have at least one ObjectClass instance in common,

- Creating equivalent binary ExclusionRule instances for each ObjectClass instance in common, and

- Rejecting those ExclusionRule instances which would match any of the observed StateDescription instances.

If the actor and actee of the RelationClass instance are members of the same ObjectClass instance, then a singleton ExclusionRule instance is created, i.e., one with a single antecedent RelationVariable instance. For example, the blocks-world relationship `on` between two `Block`-instances would result in the creation of the singleton ExclusionRule instance:

```
IF [on ?Block1 ?Block1] THEN INVALID.
```

Pairs of RelationClass instances which do not have an ObjectClass instance in common are independent of one another, and equivalent constraints do not need to be constructed.

By default, a newly-created ExclusionRule instance is marked USEABLE, i.e., it can be used in subsequent steps of the POI algorithm. If a matching situation is found in any of the observed StateDescription instances, then the marking is changed to UNUSEABLE. In essence, the POI algorithm assumes as its default that all possible ExclusionRule instances hold. The observed StateDescription instances are used as positive training instances to determine which of the defaults are inapplicable.

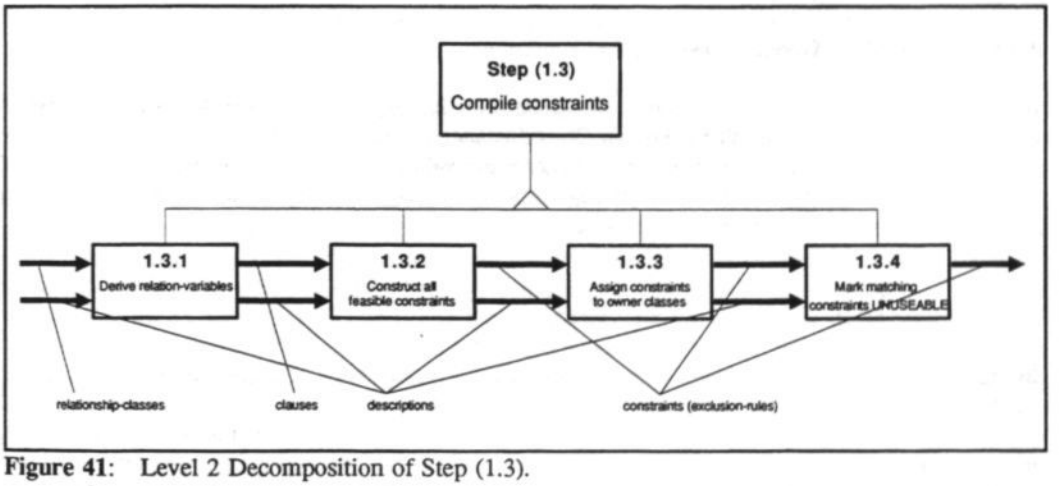

Figure 41: Level 2 Decomposition of Step (1.3).

As shown in Figure 41, the processing done in Step (1.3) can be decomposed further. In Step (1.3.1), RelationVariable instances are derived from RelationClass instances, by replacing the associated ObjectClass instances with variables formed from the ObjectClass instance's name. For example, the ObjectClass instance named `finger1` could be replaced by the variable `?Finger1`.

In Step (1.3.2), all pairs of RelationVariable instances are formed such that one the variable - the pivot variable - is common to the instance-pair. For example, if the RelationVariable `pressing ?Finger1 ?Key1` was being paired with itself, then two valid pairings would be generated: one

where `?Finger1` was the pivot, and the other where `?Key1` was the pivot. Other variables are renumbered as necessary to distinguish them from the pivot and from each other. For example, the valid pairings of `pressing ?Finger1 ?Key1` would be:

```
pressing ?Finger1 ?Key1 with pressing ?Finger1 ?Key2
```

and:

```
pressing ?Finger1 ?Key1 with pressing ?Finger2 ?Key1.
```

For each such valid pairing, an ExclusionRule instance is created with the two RelationVariable instances as its antecedents. The sole consequent of the ExclusionRule instance is the reserved word INVALID. The *useable* attribute of the newly-created ExclusionRule instance defaults to TRUE, marking it as USEABLE.

In Step (1.3.3), each newly-created ExclusionRule instance is assigned to its *owner*, i.e., the ObjectClass instance corresponding to the ExclusionRule instance's pivot variable. Assignment is done by setting the values of appropriate attributes of the ExclusionRule and ObjectClass instances.

In Step (1.3.4), each newly-created ExclusionRule instance is forward-chained against the descriptions attribute of each StateDescription instance. If the ExclusionRule instance triggers for any StateDescription instance, then the ExclusionRule instance's *useable* attribute is set to FALSE.

On completion of Step (1.3), the POI algorithm has acquired information equivalent to the relationship mappings in an entity-relationship model.

3.3.5 Step (2.1): Generate Description Language

In Step (2.1) all elementary statements in the description language are generated by instantiating each RelationClass instance with all the known ObjectInstance instances which match its actor and actee ObjectClass instances. These elementary statements are validated by testing for non-contradiction of the ExclusionRule instances. Note that the description language will be the same datatype as, but a superset of, the set of RelationInstance instances.

3.3.6 Step (2.2): Construct Version Space

In Step (2.2) the description language is used to build a version space of LatticeNode instances, with candidates being eliminated using the ExclusionRule instances. Unlike Mitchell's (1982) original algorithm, the POI algorithm builds the version space as a lattice starting from the bottom LatticeNode instance (i.e., with the description of `[ ]`). Statements from the description language are added to a given parent LatticeNode instance to form its child LatticeNode instances. Each candidate child LatticeNode instance is tested against the ExclusionRule instances. If the candidate matches any ExclusionRule instance, then it is eliminated. In Mitchell's terminology, eliminated candidates are in the Specialisation (S) set, and candidates which survive testing are in the Generalisation (G) set. Version space construction terminates when there are no more parent LatticeNode instances with untested child LatticeNode instances. Construction can be performed breadth- or depth-first. Although the baseline POI algorithm uses exhaustive breadth-first search, the POI implementations permit the search strategy to be user-selected.

3.3.7 Step (2.3): Identify States from Nodes

In Step (2.3) the StateInstance instances are extracted from the version space. In the POI algorithm, a StateInstance instance is created for each completely-described version-space LatticeNode instance which is on the *S-G frontier*, i.e., which is itself in the Generalisation set but whose children are all in the Specialisation set. LatticeNode instances whose *children* attribute is empty are tested for completeness of description by polling the ObjectInstance instances mentioned in the LatticeNode instance's descriptions. Each ObjectInstance instance checks that, of every RelationClass instance which its parent ObjectClass instance can express, either a primary or an inverse RelationInstance instance is present in the LatticeNode instance's descriptions. If not, then the LatticeNode instance is incompletely described, and no corresponding StateInstance instance is created.

3.3.8 Step (2.4): Determine Valid Transitions

In Step (2.4) the set of valid TransitionInstance instances between the StateInstance instances is determined using one of the meta-heuristics. Each meta-heuristic has two parts. For example, the SA/SSC meta-heuristic consists of a Single-Actor (SA) test, and a Single-State-Change (SSC) test. All pairs of non-identical StateInstance instances are generated. The actor ObjectInstance instances in common between the StateInstance instances are identified. If there is not exactly one actor ObjectInstance instance in common, then the potential transition is invalidated. For those potential transitions surviving this SA test, the actor's states (in terms of the descriptions attributes of the pair of StateInstance instances) before and after the potential transition are examined. If there is not exactly one of the actor's RelationInstance instances which has changed, then the potential transition is invalidated. Those potential transitions surviving both parts of the SA/SSC test are considered valid, and result in the creation of TransitionInstance instances. The SA test may be relaxed to a Multi-Actor (MA) test, and/or the SSC test may be relaxed to a Multi-State-Change (MSC) test.

3.3.9 Step (2.5): Generalise States and Transitions

In Step (2.5) the StateInstance and TransitionInstance instances are generalised to become StateClass and TransitionClass instances, respectively. As shown in Figure 42, the processing done in Step (2.5) can be decomposed further. In Step (2.5.1), StateClass instances are generalised from a list of StateInstance instances. In the POI algorithm the StateInstance instances are added to a list of instances to be generalised. The first item on the list is generalised to become a StateClass instance by replacing the name of each ObjectInstance instance in the description with the name of a variable formed from the name of the corresponding ObjectClass instance. The list is scanned to remove other StateInstance instances which also match the newly-generalised StateClass instance. The process repeats until the list is exhausted.

In Step (2.5.2), TransitionClass instances are generalised from a list of TransitionInstance instances in a similar way. The first item on a list of TransitionInstance instances is generalised by finding the StateClass instances corresponding to TransitionInstance instance's StateInstance instances. The list of TransitionInstance instances is scanned to remove those instances which share the same pair of StateClass instances. The process repeats until the list is exhausted.

In Step (2.5.3), the newly-created StateClass and TransitionClass instances are cross-referred with one another. This sub-process may involve renegotiation of the descriptions of one or more StateClass instances to ensure that the descriptions of the cross-referenced TransitionClass instances are valid.

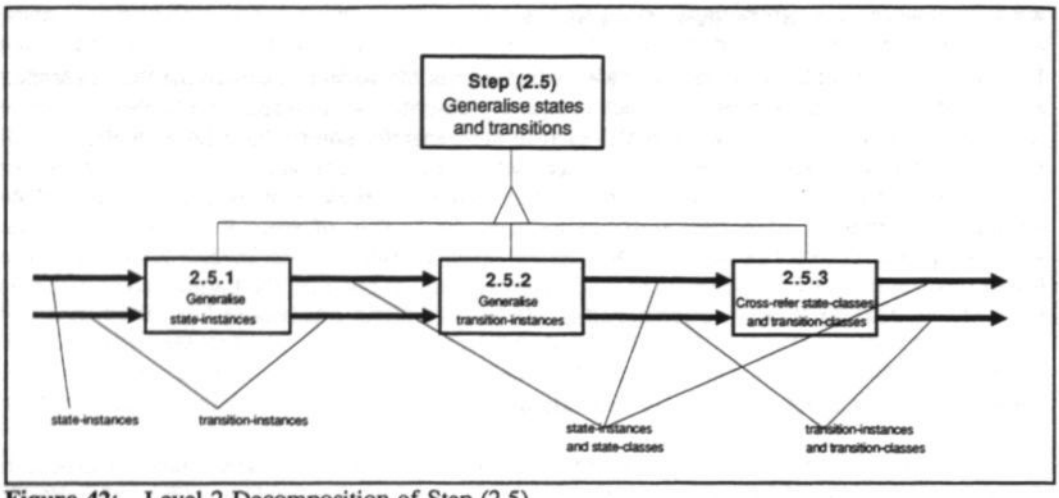

Figure 42: Level 2 Decomposition of Step (2.5).

3.3.10 Step (2.6): Format as Operators

In Step (2.6) the TransitionClass instances - which are equivalent to preconditions- and effects-lists - are reformatted as STRIPS operators, with preconditions-, add-, and delete-lists. The STRIPS format is obtained by performing set-subtraction between the TransitionClass instance's preconditions- and effects-lists, i.e., between the descriptions of its "from" and "to" StateClass instances. For each TransitionClass instance, two PlanningOperator instances are created. One PlanningOperator instance's add-list consists of those RelationInstance instances which appear in the TransitionClass instance's effects-list but not in the TransitionClass instance's preconditions-list. The PlanningOperator instance's delete-list consists of those relationships which appear in the TransitionClass instance's preconditions-list but not in the TransitionClass instance's effects-list. The PlanningOperator instance's preconditions-list consists of those RelationInstance instances which appear in both lists in the TransitionClass instance, i.e., the filter preconditions. ObjectClass membership tests are added to the PlanningOperator instance's preconditions-list. The second PlanningOperator instance is a copy of the first PlanningOperator instance, but with the values of the add-list and delete-list attributes exchanged. Finally, each newly-created PlanningOperator instance is given a unique name.

3.4 IMPLEMENTATION

There are two implementations of the POI ontology and algorithm: a single-agent implementation and a multi-agent implementation.

3.4.1 Single-Agent Implementation

The single-agent implementation is known as the Dutch Utilisation Centre Activity Scheduling System (DUC-ASS). DUC-ASS development was part of a larger project to build a pilot Dutch Utilisation

Centre[36], partly funded by the Nederlands Instituut voor Vliegtuigontwikkeling en Ruimtevaart (the Netherlands Agency for Aerospace Programs). A requirement was that the DUC-ASS was designed to support the design of spacecraft payloads, the generation of payload operating procedures, the planning and scheduling of payload operations, the control of the schedule execution under nominal and non-nominal conditions, and the post-mission comparison of achieved against scheduled activities. More details are available in (Grant, 1992c). This thesis is concerned specifically with the DUC-ASS facilities for supporting payload design.

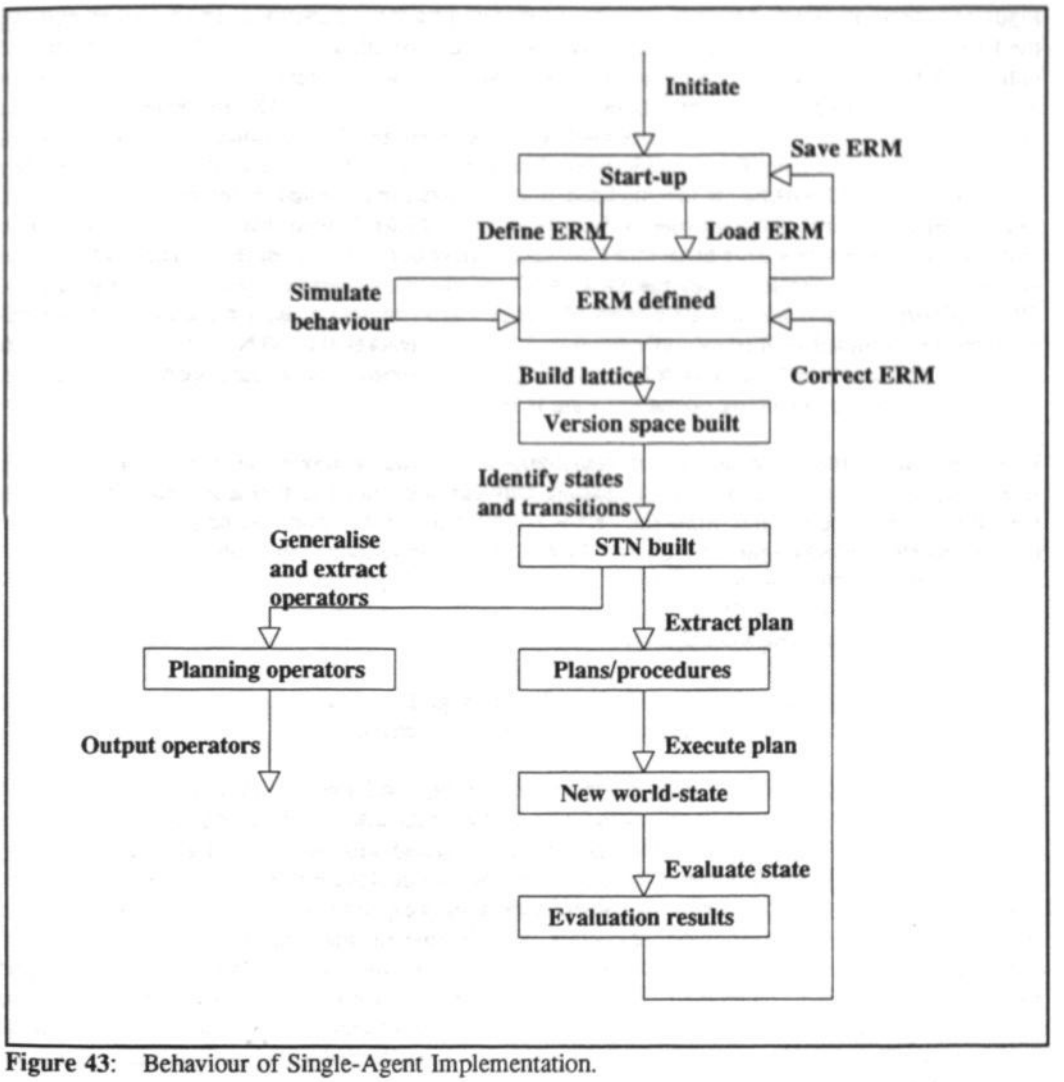

Figure 43: Behaviour of Single-Agent Implementation.

[36] The Dutch Utilisation Centre will support users of Dutch payloads on spacecraft and sounding rockets. DUC-ASS development has been done in consultation with a representative user from the Space Research Organisation Netherlands (SRON). Informal feedback on the DUC-ASS's functionality has also been obtained from the Nationaal Lucht- en Ruimtevaart Laboratorium (The Netherlands' National Aerospace Laboratory) and from the European Space Operations Centre (ESOC), Darmstadt, Germany.

DUC-ASS implements the full POI ontology and Part 2 of the POI algorithm, i.e., Induction. Payload design is supported by Steps (2.1) to (2.4). Steps (2.5) and (2.6) are needed only for the purposes of my research. Paralleling Steps (2.5) and (2.6), additional functionality has been implemented to support payload operating plan/procedure generation from the state-transition network resulting from Steps (2.3) and (2.4). This parallel functionality was necessary to meet the requirements of the Nederlands Instituut voor Vliegtuigontwikkeling en Ruimtevaart. We are concerned here solely with the production of planning operators, and not with the parallel plan/procedure-generation functionality.

Figure 43 shows the DUC-ASS's behaviour in the form of a state-transition network. After initiating the DUC-ASS, it is in the "Start-up" state, with no domain-specific information. The user can choose either to define a new domain *ab initio* ("Define ERM") or to load a previously-defined domain from a disk file ("Load ERM"). The user defines a domain by instructing DUC-ASS to create object-classes and relationship-classes, and then to instantiate the object-classes. Whenever a new relationship-class is created, DUC-ASS identifies possible interrelationship constraints and asks the user whether they apply (i.e., are USEABLE) in the intended domain. After the complete set of domain objects, relationships and constraints has been defined (shown as "ERM defined" state), the user can choose either to simulate the domain's behaviour ("Simulate behaviour"), or to identify domain states ("Build lattice"). The latter choice causes the DUC-ASS to induce the version space. After reaching the "Version space built" state, the DUC-ASS identifies valid transitions using the user-selected meta-heuristic. On completion of transition validation, DUC-ASS reaches the "STN built" state from which planning operators may be extracted by traversing the "Generalise and extract operators" transition. After extraction, the "Planning operators" state is reached.

Figure 43 also shows the additional functionality to support procedure generation, planning, scheduling, control, and post-mission evaluation. This additional functionality is accessed by traversing the "Extract plan" transition from the "STN built" state. Post-mission evaluation may result in corrections to the domain model ("Correct ERM"), requiring re-induction of the STN.

3.4.2 Multi-Agent Implementation

The multi-agent implementation is known as the Message-Based Architecture (MBA) testbed. The MBA testbed was developed solely for the purposes of my research.

The MBA testbed implements most of the POI ontology and the full POI algorithm. The POI algorithm is embedded in the Induction Module, which is made available to all Agents that are to have a POI capability (*POIAgents*). The POI algorithm is integrated with reactive and generative planning (Grant, 1991). Integration is achieved by implementing an inheritance hierarchy of functionalities, in which POIAgents also have reactive and generative planning capabilities. At the root of the inheritance hierarchy is the NonIntentionalAgent class, which implements the basic functionalities of communicating with other agents, of maintaining state as a set of relationships with other agents, and of updating state and generating outgoing messages in response to incoming messages. The ModellingAgent class is the specialisation of the NonIntentionalAgent class that adds reactive planning. The PlanningAgent class is the specialisation of the ModellingAgent class that adds generative planning. Finally, the POIAgent class is the specialisation of the PlanningAgent class that adds the POI capability.

Use of the testbed can be characterised as a simulation of the interactions between a multi-agent problem domain and one or more POIAgents. First, the testbed user defines the problem domain in terms of domain objects, modelled as a set of NonIntentionalAgents. Second, the user creates one or more POIAgents. On creation, the POIAgents have no knowledge of the problem domain. The user

triggers interaction between the POIAgents and the problem domain by giving the POIAgents a series of problem-domain goals to achieve. The POIAgents attempt to achieve the goals by acting on the agents in the problem domain. In so doing, the POIAgents observe different states of the problem domain. These observed states are the input information for the full POI algorithm. The POIAgents gain knowledge about the problem domain by converting the observed states into a domain model (i.e., using Part (1) of the POI algorithm) and then inducing planning operators from this domain model (i.e., using Part (2) of the POI algorithm). The POIAgents can then use the newly-induced planning operators in generating a plan. Execution of the generated plan achieves the user's goals. The POIAgents report their success to the user. More details of the MBA testbed can be found in (Grant, 1992b).

3.4.3 Implementation and Experimental Environments

DUC-ASS and the MBA testbed have been implemented in the Smalltalk/V 286 object-oriented programming language (Digitalk, 1988), on IBM-compatible PCs running MS-DOS. The development machine was a 80386-class PC, running at 20 MHz, with 4 MB memory and a Norton Computing Index of 17.6. Both implementations have been ported successfully to a variety of PCs. Most of the DUC-ASS runs documented in Chapter 4 were run on a 80486-based PC, running at 50 MHz, with 8 MB memory and a Norton Computing Index of 77.8. Other PCs with 16 MB and 40 MB memory were also available, if necessary. All the MBA testbed runs, documented in Chapter 5, were performed on the development machine.

One implication of the choice of Smalltalk/V 286 is that it runs in the 80x86 processor's protected mode. This mode is limited in addressing capabilities to a maximum of 16 MB memory. The limitation represents a ceiling on the size of domain for which experiments could be performed. To escape from the ceiling, the DUC-ASS and MBA testbed programs would have to be ported to a later version of Smalltalk/V, but this would have required too much time and effort. As described in Chapter 4, the memory ceiling made it impossible to induce operators for the POI meta-domain. The other experiments had requirements below this ceiling, enabling my thesis claims to be substantiated.

3.4.4 Implementing the Ontology

In principle, the ontology has been implemented by creating a Smalltalk/V class for each ontological class. In practice, there are some divergences. For DUC-ASS, the divergences are slight, e.g., there are no State and Transition superclasses, the functionality of Heuristic is coded into a ClassOwner superclass (to enable version-space partitioning), and the ontological classes ObjectClass, ObjectVariable, and ObjectInstance have been renamed EntityClass, EntityVariable, and EntityInstance, respectively, to avoid a name-clash with Smalltalk/V's class-library. DUC-ASS extends the ontology in that RelationClass is specialised into the object-classes PrimaryRelationClass and InverseRelationClass. The divergences are greater for the MBA testbed, because it was developed much earlier. Many of the ontological classes have not been implemented, notably the Transition sub-hierarchy. The ObjectInstance functionality is a part of the NonIntentionalAgent class.

The mappings of ontological classes to implemented Smalltalk/V classes are shown in Figure 44. These Smalltalk/V classes are at the leaves of a larger class hierarchy. Additional Smalltalk/V classes were needed to manage the ontology, to provide forward-chaining inference, or to serve as superclasses (rationalising the common attributes or operations of the leaf-node object-classes) or as auxiliary functionality. The complete object-class inheritance hierarchy in the DUC-ASS application is shown in Figure 45. The DUC-ASS object-class is the root of the application-specific part of the Smalltalk/V

class-hierarchy. The inheritance hierarchy of the MBA testbed is not shown here because it diverges from the POI ontology, but details have been given in (Grant, 1992b).

Ontological Class	DUC-ASS	MBA
Structural model:		
Object	Entity	-
ObjectClass	EntityClass	DomainClass
ObjectInstance	EntityInstance	NonIntentionalAgent (and its specialisations: ModellingAgent, PlanningAgent, and POIAgent).
ObjectVariable	EntityVariable	-
Relation	Relation	-
RelationClass	RelationClass (together with its subclasses: PrimaryRelationClass and InverseRelationClass)	-
RelationInstance	RelationInstance	Relationship
RelationVariable	RelationVariable	Clause
ExclusionRule	ExclusionRule	Rule
StateDescription	-	StateDescription
Behavioural model:		
State	-	-
StateClass	StateClass	-
StateInstance	StateInstance	(also StateDescription)
Transition	-	-
TransitionClass	TransitionClass	-
TransitionInstance	TransitionInstance	-
Heuristic	-	-
PlanningOperator	PlanningOperator	Activity
Linking model:		
LatticeNode	LatticeNode	WorldState

Figure 44: Mapping of POI Ontology onto Smalltalk Classes.

Domains are modelled by instantiating the Smalltalk/V classes. Each domain object-class is modelled as an instance of the EntityClass class. The instance takes the name of the domain object-class. For example, the `Finger` class of objects in the piano-playing domain would be modelled in DUC-ASS as an instance of the EntityClass class with the name "Finger". Similarly, each object-instance in the domain is modelled as an instance of the EntityInstance class. For example, the `finger2` domain object would be modelled as an EntityInstance instance with the name "finger2". In the same way, the `pressing` relationship-class would be modelled as a RelationClass instance named "Pressing", and

the relationship-instance `pressing finger2 key3` would be modelled as a RelationInstance instance named "pressing" with the values of its actor and actee attributes pointing, respectively, to the "finger2" and the "key3" ObjectInstance instances.

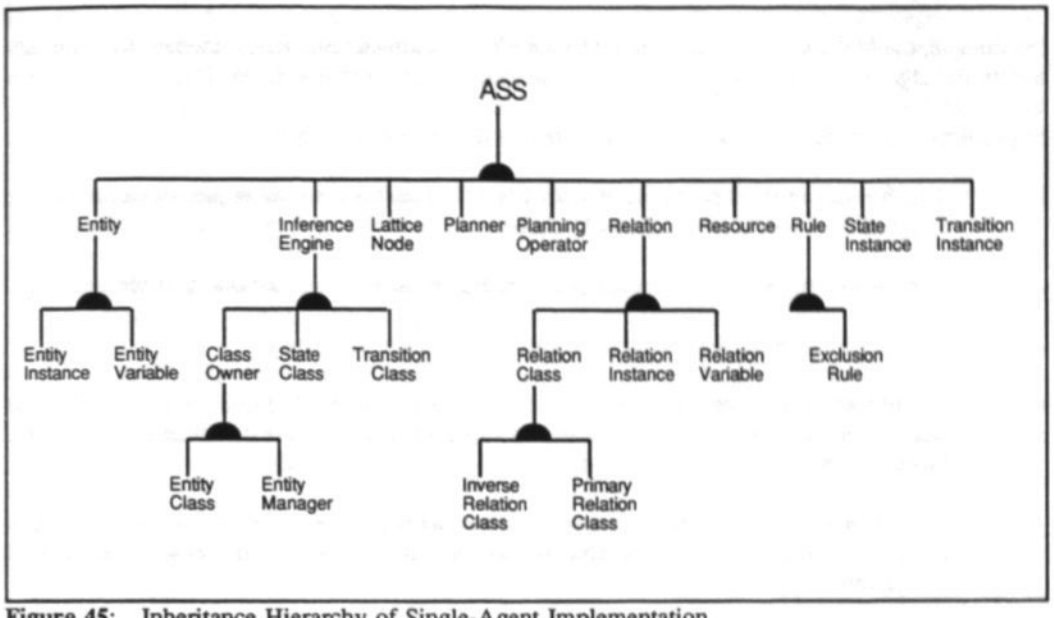

Figure 45: Inheritance Hierarchy of Single-Agent Implementation.

The Smalltalk/V language provides an object-attribute-value representation; no representation is provided for inter-object relationships. Relationships in the POI ontology are implemented by means of attributes. For example, the `member-of` relationship in the structural model is implemented by providing the Smalltalk/V EntityInstance class with an attribute named *myClass* and the Smalltalk/V EntityClass class with an attribute named *instances*. In essence, such attributes designate the *roles* of the relationship, in the sense intended by Chen (1976). The class-instance relationship in the piano-playing domain between the `Finger` object-class and its `finger2` instance would be implemented by:

- appending a pointer to the EntityInstance instance named `finger2` to the value of the *instances* attribute of the EntityClass instance named `Finger`, and
- setting the value of the *myClass* attribute of the EntityInstance instance named `finger2` to a pointer to the EntityClass instance named `Finger`.

The Smalltalk/V source code is simpler than the above description. The following code would appear in the EntityClass operation which creates the EntityInstance instance[37]:

```
instances add: anInstance.
anInstance myClass: self.
```

The first line causes `anInstance` to be added to the EntityClass instance's *instances* attribute. The

[37] `self` refers to the EntityClass instance (e.g., `Finger`) and `anInstance` points to the EntityInstance instance (e.g., `finger2`).

second line instructs `anInstance` to set its *myClass* attribute to point to the EntityClass instance.

3.4.5 Implementing the Algorithm

In principle, the POI algorithm has been implemented by creating a Smalltalk/V method for each sub-step in the algorithm. The mapping of steps to methods is shown in Figure 38.

In practice, implementation has been complicated by the following issues:

- The POI algorithm is presented at a single level of abstraction, but implementation must be assigned to a variety of Smalltalk/V classes.
- Various orderings of the steps are possible. Figure 38 presents just one feasible ordering.
- The implementation has had to be refined to maximise efficiency.
- The implementation has been enhanced beyond the baseline POI algorithm described in this chapter, in that it incorporates a number of countermeasures to the combinatorial explosion. More details are given in Section 3.6.
- Neither of the two implementations emphasises the two Parts of the POI algorithm equally. DUC-ASS emphasises Part 2 (Induction), and the MBA testbed emphasises Part 1 (Acquisition).

```
!EntityManager methods !

runGlobal

  "Purpose:
   EntityManager performs global-STN run, with printStats.

   Update history:
    2 Jul 95 - created from runExperiment (TJG)."

   myUI nl; answer: 'Running global at ',(Time now printString),
                    ' on ',(Date today printString),
                    ' with:'.
   subordinates do: [:aClass |
      myUI answer: '    ',(aClass myName),
                   ': ',(aClass instances size printString),
                   ' instances.'].
   self generateLanguage21.
   self buildLattice22.
   self identifyStates23.
   self determineTransitions24.
   self extractStateClasses251.
   self extractTransitionClasses252.
   self extractPlanningOperators26.
   self printStats.
   myUI nl; answer: 'Global run completed at ',(Time now printString),
                    ' on ',(Date today printString),' with:'.
   ^subordinates do: [:aClass |
       myUI answer: '    ',(aClass myName),
                    ': ',(aClass instances size printString),
                    ' instances.']! !
```

Figure 46: Smalltalk/V Source Code of runGlobal method.

DUC-ASS follows the POI algorithm most faithfully in implementing Part 2. All the Smalltalk/V methods listed in Figure 38 are implemented in the ClassOwner class. These methods are inherited by EntityClass for the purposes of inducing class- and instance-states, and by EntityManager for the purposes of inducing global-states. For user-convenience, the methods that implement the POI algorithm are called in sequence by one of a small number of higher-level EntityManager methods. The method-names indicate which step of the POI algorithm they implement, e.g., the method named `extractStateClasses251` implements POI Step (2.5.1). Figure 46 shows the Smalltalk/V source code of the `runGlobal` method, which induces operators via global states. Similar methods named `runClass` and `runInstance` induce operators via class- and instance-states, respectively.

Part 1 (Acquisition) is implemented fully in the MBA testbed, but in a different order to that presented in Figure 38. In effect, the steps are executed as follows:

- *Step (1.1).* An instance of the Definer class acquires StateDescriptions from the user by means of its `defineMessages` method.

- *Step (1.2.4).* The Definer identifies ObjectClasses from the user-provided StateDescriptions in the `createClassesFrom: messages` method, implemented in its ModellingAgent superclass. All subsequent steps of Part 1 of the POI algorithm are then delegated to the newly-created ObjectClasses by sending them a `perceive: messages` message.

- *Steps (1.2.1) and (1.2.2).* Each ObjectClass extracts the subset of messages relevant to itself, and, from this subset, identifies RelationInstances and EntityInstances by calling its own `extractRelationsFrom:` and `extractInstancesFrom:` methods.

- *Steps (1.3.2) to (1.3.4).* In the course of identifying RelationInstances, the ObjectClass delegates the compilation of ExclusionRules to the newly-created RelationInstances by sending them a `createExclusionsWith:for:unlessIn:` message.

- *Step (1.3.1).* Newly-created ExclusionRules instantiate their antecedents (i.e., RelationVariables) from a text-string by sending themselves a `setTextTo:` message.

Step (1.2.3) is not implemented, because the MBA testbed does not implement the RelationClass.

Both implementations use the same, single-window, character-based user interface, as shown in the screendump in Figure 47. The full-screen window is divided into four panes. The large pane occupying two-thirds of the screen on the left-hand side provides an output area where text can be displayed using a teletype-like presentation. The three smaller panes on the right-hand side enable the user to make inputs. The upper pane lists those Smalltalk/V classes to which the user is permitted to send inputs. Using the mouse, the user can select one of the listed classes by moving the cursor over it, and clicking the left button. The middle pane lists the instances of the Smalltalk/V class last selected in the upper pane. In a similar way, the user can select one of the listed instances. The lower pane lists those messages which the user can send to the instance last selected in the middle pane. During execution of the message, the recipient instance causes text to be displayed in the large output pane.

Figure 47: DUC-ASS Screendump - After Loading Domain Model.

Figure 47 shows the situation immediately after the user has loaded the Piano-Playing domain model and created two instances of each of the Finger and PianoKey ObjectClasses. The user would initiate the induction of planning operators from a global STN by clicking on the `runGlobal` message, visible in the lower input pane on the right-hand side. A few of the alternative actions open to the user at this point include:

- inducing a set of planning operators from class STNs, using `runClass`.
- inducing a set of planning operators from instance STNs, using `runInstance`.
- inducing three sets of planning operators, from global, class, and instance STNs, using `runExperiment`.
- inducing the domain's StateInstances (i.e., Steps (2.1) to (2.3)), marking some of the StateInstances as UNUSEABLE, and then completing the rest of the POI algorithm (i.e., Steps (2.4) to (2.6)). The user would do this by first selecting `run21To23`. When induction of the StateInstances was finished, the user would select the StateInstance class in the upper pane, select each StateInstance instance in turn in the middle pane, sending each selected instance the `makeUnuseable` message from the lower pane. Finally, the user would re-select the EntityManager class and its single instance, and send it the `run24On` message to complete the rest of the POI algorithm.

Figure 48 shows the statistics collected at the end of a run.

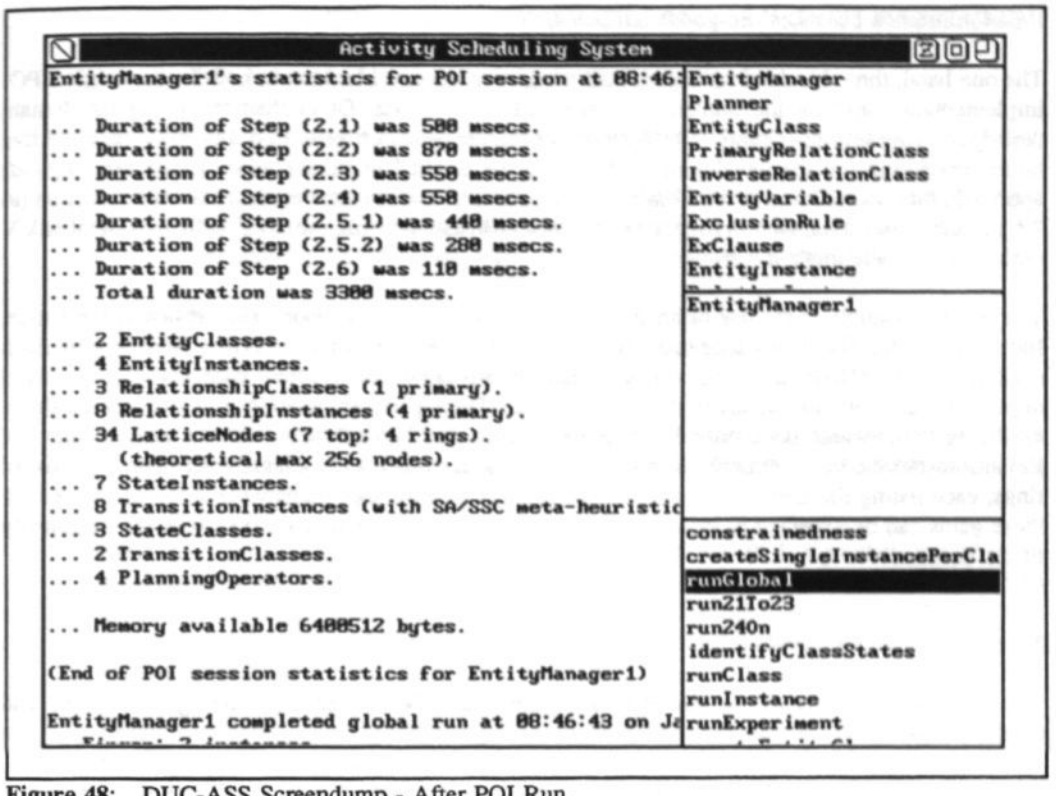

Figure 48: DUC-ASS Screendump - After POI Run.

Each of the four panes has an associated pane-menu, although those for the upper and lower panes on the right-hand side are dummies. A pane menu may be popped-up by moving the cursor over the pane and clicking the right button. The user can save the session trace to an ASCII file by selecting `save` from the output-pane's menu.

3.4.6 Refinements made during Implementation

Several refinements were made during the course of implementation. Most of the refinements were aimed at improving the algorithm's efficiency, both in terms of memory usage and of runtime. Two example refinements are outlined here.

The first example is the application of *no-goods*, by analogy with reason maintenance. As described in this thesis, candidate elimination occurs every time a new candidate LatticeNode instance is created by forwards-chaining each of the ExclusionRule instances. During implementation, it was found that this approach was wasteful of processing power. Accordingly, Step (2.2) was modified so that, prior to version space construction, inferencing was performed for all pairs of RelationInstance instances. If any pair caused an ExclusionRule instance to trigger, then each RelationInstance instance added the name of the other RelationInstance instance to its *no-goods* attribute. Then, to determine the validity of a candidate LatticeNode instance during version space construction, it was sufficient to consult the *no-goods* attributes of the parent LatticeNode instance and of the RelationInstance instance being added to it. If either *no-goods* attribute listed any of the other's descriptions, then the candidate was eliminated. In essence, the candidate elimination rules are compiled into the values of the

RelationInstance instances' *no-goods* attributes.

The one-hand, three-blocks, one-table blocks world was run as a benchmark, both for the original POI implementation and for the implementation based on *no-goods*. Other than restricting the domain description language to primary relationships, no countermeasures were used. Induction required five hours processing (on the development PC) using the original implementation. Using the *no-goods* approach, this was reduced to 16 minutes on the same PC. It was observed that a further reduction (to 7.5 minutes) was obtainable by replacing the object-instance-naming method built into Smalltalk/V with a tailor-made method.

The second example is the use of an improved data-structure for indexing the version-space lattice. Initially, the ClassOwner instance maintained a list of LatticeNode instances as they were created. Each candidate LatticeNode must be checked against all previously-created LatticeNodes, to avoid duplicating the common nodes in the version space. During implementation, it was observed that a candidate LatticeNode need only be checked against other LatticeNodes with the same number of RelationInstances, i.e., LatticeNodes on the same *ring*. By making the ClassOwner maintain a list of rings, each listing the LatticeNodes on that ring, major performance gains were achieved. Moreover, these gains can be expected to increase with the depth of the version-space, i.e., with the complexity of the domain.

3.5 LIMITATIONS

The following limitations of the POI algorithm have been identified during its design and implementation:

- *Completeness*. The algorithm cannot guarantee the induction of a complete set of planning operators for an arbitrary set of world-state descriptions. For example, suppose that the POI algorithm was provided only with the following blocks-world state description:

  ```
  [[holding hand1 block1]].
  ```

  Then the POI algorithm would be able to generate (the equivalent of) the **`pickUp`** and **`putDown`** operators, but it would be unable to generate the **`stack`** or **`unstack`** operators. The POI algorithm's user is responsible for selecting a minimal set of world states to present to the POI algorithm. A minimal set would be one which allowed Part 1 of the POI algorithm to extract all object-instances, all object-classes, all relationship-classes, and all constraints for the domain concerned. The best that the POI algorithm can do is to induce knowledge that is consistent with, and complete by reference to, the examples provided by the user.

- *Finite domain*. The POI algorithm depends on the number of object-classes, numbers of instances per class, and numbers of relationship-classes between each pair of object-classes being finite.

- *Binary relationships*. The POI algorithm depends on the input set of world-state descriptions taking the form of binary relationships. Relationships such as "workpiece1 is held in lathe3 using jig4" must be expressed as conjunctions of binary relationships, e.g., "jig4 retains workpiece1 AND jig4 fitted to lathe3". This has advantages and disadvantages. The advantages (Frost, 1986, pp. 24 and 25) of binary relationships are that:

- the uniformity of the representation results in simplified storage and processing.
- it is easier to add new knowledge to the knowledge base.
- many-to-many relationships can be represented with no replication of knowledge.
- it is possible to represent higher-order relationships (e.g., "Bob thinks that Bill likes Sue") if the relationships are labelled. The POI ontology permits labelling of relationship-instances, but higher-order relationships are excluded.

and the disadvantages are that:

- more memory is generally required.
- since relationships with an object-class or object-instance in common cannot always be stored in physical proximity, retrieval of collections of related knowledge takes more effort.

Note that there is no limit on the expressiveness of the POI algorithm and that all disadvantages are implementation-related. In the worst case, the user may be required to specify an additional class of objects or of relationships in order to express his/her domain knowledge. In summary, binary relationships - if sometimes inconvenient to the user - are entirely general.

- *Binary constraints.* The algorithm depends on the ability to express invalid domain states in the form of binary constraints. The impact of this limitation is best clarified by means of an example. The entity-relationship model for the Nilsson (1980) variant of the blocks world, as presented in Chapter 2, allows blocks to assume four possible roles: as actee of the `holding` relationship, as actor and actee of `on`, and as actor of `onTable`. To express the domain constraint that blocks should not be left "floating in mid-air", an ExclusionRule instance with three antecedent RelationVariable instances would be needed, as follows:

```
IF [notHeld ?Block1]
AND [notOn ?Block1]
AND [notOnTable ?Block1]
THEN INVALID.
```

This is a triple constraint, which cannot be represented using the POI ontology. One solution would be to impose the requirement that the user must model the domain in such a way that no domain object-class takes part in more than two relationship-classes. This can be done by introducing inheritance in the domain's object-class hierarchy. More details are given in Section 4.10.

- *Non-metric relationships.* The algorithm is limited to non-metric relationships. In particular, temporal relationships cannot be supported. Extending the POI ontology and algorithm to metric dimensions and relationships is outside the scope of my research, and would be an area suitable for study by other researchers in future.

- *Combinatorial explosion.* Most seriously, the algorithm is inherently combinatorially explosive. This is a consequence of basing it on the version space and candidate elimination algorithm. The latter algorithm was designed for (single) concept learning, and is known to be problematic for continuous attribute spaces (Haussler, 1988), disjunctive target concepts

(Murray, 1987), and noisy example data (Kalkanis and Conroy, 1991). In particular, planning domains necessarily exhibit multiple disjunctive target concepts, i.e., StateInstances.

In summary, the POI algorithm is limited to non-metric, binary relationships and to binary constraints, and it cannot guarantee to induce a complete operator-set for an arbitrary set of inputs. The most serious limitation is that the algorithm is inherently combinatorially explosive. The latter limitation is addressed in more detail in the following section.

3.6 COMBATTING COMBINATORIAL EXPLOSION

This section describes the actions taken to combat the combinatorial explosion the POI algorithm inherits from the version space and candidate elimination algorithm. Such actions will be termed *countermeasures.*

The complexity of the POI algorithm was analyzed theoretically to confirm that there was a combinatorial explosion and to show where attention was needed for efficiency of implementation. Then, a set of countermeasures was selected. The experience in implementing and using these countermeasures is described in Chapter 4.

3.6.1 Theoretical Complexity Analysis

This section analyzes the theoretical complexity of the POI algorithm.

In POI, domains are modelled in terms of object-classes, relationship-classes, and constraints. Changing one of these inputs changes not only the essential nature of the domain, but also the runtime and memory requirements of the POI algorithm. In other words, the POI algorithm's complexity must be analysed across a variety of domains. I term this *across-domain* complexity.

In principle, across-domain complexity could be assessed by varying each of the inputs in turn. In practice, the constraints are dependent on the object-classes and relationship-classes. Moreover, relationship-classes are largely determined by the object-classes. Since the number of object-classes is the key, assessment has been confined to varying the number of object-classes.

There is another input: the number of object-instances in each object-class. Changing the number of object-instances does not change the domain's essential nature. For example, a three-blocks world shares with a two-blocks world the characteristic behaviour of stacking blocks. However, changing the number of object-instances does influence the size of the state-space. In Nilsson's (1980) representation, a three-blocks world has 22 valid states, but a two-blocks world has only five. Inducing a state-space with 22 states requires runtime and memory than inducing a space containing only five states. Therefore, the POI algorithm's complexity must also be analysed in changing the number of object-instances within a given domain. I term this *within-domain* complexity.

Within-domain complexity can be readily assessed by:

- keeping the object-classes, relationship-classes, and constraints constant, and
- varying the number of object-instances for each object-class.

Step	Complexity	Remarks
1.1	$O(d)$	
1.2	$O(d)$	
1.3	$O(d * r^2)$	
2.1	$O(r * i^2)$	
2.2	2^c	Worst case, i.e., when there are no constraints
2.3	$O(s)$	
2.4	$O(s^2)$	
2.5	$O(s^2)$	
2.6	$O(t)$	

Figure 49: Theoretical Complexity Analysis of POI Algorithm.

Figure 49 shows an analysis of the worst-case complexity for each step in the POI algorithm, where:

d is the number of StateDescription instances,
r is the number of RelationClass instances,
i is the number of ObjectInstance instances,
c is the number of statements in the domain description language,
s is the number of StateInstance instances, and
t is the number of TransitionClass instances.

The relationships between these variables are specific to the application domain being modelled.

Further analysis concentrates on the step with the highest complexity, Step (2.2). The worst case for version-space construction is when there are no domain constraints. In that case, the version space would be the same as the rule space. The size of the rule-space depends on the size of the domain description language, i.e., on the size of the parameter c. The domain description language is the set of RelationInstance instances, which are themselves dependent on the numbers of ObjectClasses and RelationClasses. There are two types of RelationClass:

- *A RelationClass that joins an ObjectClass J to itself.* Such a RelationClass contributes $(i_J^2 + 2i_J)$ RelationInstances to the domain description language, where i_J is the number of ObjectInstances of the ObjectClass J.

- *A RelationClass that joins different ObjectClasses J and K.* Such a RelationClass contributes $(i_J.i_K + i_J + i_K)$ RelationInstances to the domain description language, where i_J and i_K are the numbers of ObjectInstances of the ObjectClasses J and K.

The parameter c is the sum of the contributions of all the RelationClasses. The theoretical maximum size of the version-space is then 2 to the power of c. Figure 50 lists the size of the rule space for the domains considered in my research[38]. For simplicity of comparison, each domain is assumed to have

[38] The domains are described in Chapter 4.

just one instance of each object-class[39].

Domain	Value of c
Finger-crossing	0
Piano-playing	3
Blocks World (Genesereth and Nilsson, 1987)	5
Tank-farm	6
Blocks World (Nilsson, 1980)	8
Blocks World (Ramsay and Barratt, 1991)	14
Dining philosophers problem	15
Aircraft scheduling	33
POI	60
High Performance Capillary Electrophoresis	87

Figure 50: Size of Rule Space for Various Domains.

Since domains are invariably constrained, the version space will generally not reach its maximum size. The size of the actually-constructed version space divided by the size of the rule space will be termed the *figure of merit*. The larger the figure of merit is, the better. The figure of merit is easiest assessed by performing POI runs, i.e., empirically. Sections 4.4 and 4.5, respectively, report the results of an empirical analysis of within- and across-domain complexity.

3.6.2 Potential Countermeasures

In developing potential countermeasures, attention was focused on the efficiency of version-space construction. Several potential countermeasures were not investigated, for the following reasons:

- *Domain-specific heuristics*. The use of domain-specific heuristics was rejected because it would make the POI algorithm domain-dependent. Moreover, the issue of acquiring the heuristics would then arise.

- *Plan-space lattice*. Version-space construction in plan-space, instead of state-space, was rejected on the grounds that the size of plan-space is of the order of the square of the size of the state-space for the same domain. Moreover, a plan-recognition capability would be needed in Part 1 of the algorithm; plan-recognition is itself an active area of research.

- *Class-level version space*. The use of RelationClass instances as the basis for version space construction was rejected when it was observed that the resulting space would no longer exhibit the subsumption property.

---

[39] In practice, the finger-crossing, dining philosophers, POI, and High Performance Capillary Electrophoresis domains would have to have more than one instance of certain object-classes in order to be meaningful. For example, finger-crossing is not possible with only one finger.

- *Partial version space.* It might be possible to induce a full operator-set from a partly-constructed version-space. One such approach (using depth-first search, followed by beam-search) was implemented. However, as this countermeasure no longer guarantees the completeness of the induced state-transition network, I decided not to investigate it further.

The countermeasures investigated and implemented fully are as follows:

- *Introducing ObjectClass inheritance.* A facility was introduced to enable ObjectClasses to be placed in a generalisation-specialisation (inheritance) hierarchy. The original motivation was to be able to represent the HPCE domain. It was observed that inheritance had several other benefits. Most importantly for complexity, it reduced the number of RelationClass instances. This lead to a reduction in the domain description language, so reducing the size of the version space. Another valuable benefit was that inheritance could be used to avoid the need to express constraints with an arity greater than two.

- *Version-space partitioning.* When the complete domain description language is used for version-space construction, a State-Transition Network containing world-state descriptions is obtained. Such a State-Transition Network will be termed a *global STN*. A global STN contains *global states* and *global transitions*, i.e., states and transitions involving all the objects in the domain. By contrast, the version space can be partitioned by limiting induction to a subset of the ObjectClass, ObjectInstance, RelationClass and/or RelationInstance instances. By partitioning, each subset can induced separately, yielding a *sub-STN*[40]. The set of sub-STNs can then be merged to reproduce the global STN. This has the advantage that the complexity of Step (2.2) is reduced. Version-space partitioning has been successfully implemented, and has been investigated fully for partitioning by ObjectClasses (yielding a *class-STN* containing *class-states* and *class-transitions*) and by ObjectInstances (yielding an *instance-STN* containing *instance-states* and *instance-transitions*).

The experience in introducing inheritance suggested another potential countermeasure: to introduce a whole-part (decomposition) hierarchy. Sub-domains could then be "packaged-up" as object-instances. It has not been necessary to investigate this countermeasure, e.g., for handling real-world domains. Other researchers may wish to extend the POI ontology and algorithm to introduce decomposition.

3.7 CHAPTER 3 SUMMARY

This chapter has documented the POI algorithm and its underlying ontology. Functionality, application, and tractability requirements have been compiled. The POI ontology connects a structural model of the domain to its behavioural model by means of a linking model. The ontology has been presented using Chen's (1976) entity-relationship model notation. Since the POI algorithm also takes its input in the form of a entity-relationship model of the domain, the POI ontology is a meta-representation.

Following accepted software engineering practice, the POI algorithm has been described in terms of its functional decomposition. At the top level of the decomposition hierarchy, the algorithm is decomposed into two Parts. At the second level, the algorithm consists of nine steps. Each step is described in detail, with some steps being decomposed another level.

---

[40] Sub-STNs are often encountered in software engineering. For example, many CASE tools elicit sub-STNs, rather than a global STN, from their user.

Single- and multi-agent implementations are described. The implementation-language classes are related directly to the classes of entities found in the POI ontology. The behaviour of the single-agent program is shown as a state-transition network. Key refinements made during implementation are outlined.

The limitations of the POI algorithm are summarised. The most serious limitation is that the algorithm is inherently combinatorially explosive. A theoretical analysis of the algorithm's complexity is documented. In the worst case, the runtime and memory usage would be proportional to 2 to the power of the size of the domain description language (i.e., 2 to the power of the number of RelationInstances). A number of potential countermeasures to the combinatorial explosion are described. Two of these countermeasures - object-class inheritance and version-space partitioning - have been implemented fully.

4 OPEN-LOOP EXPERIMENTS

The purpose of this chapter is to describe a programme of open-loop experiments. Open-loop testing involved supplying a set of inputs to an implementation of Part 2 (Induction) of the POI algorithm and examining its output, without feedback from output to input (see Figure 51). To ensure the domain-independence of the results, the experiments were repeated for several domains. The programme was designed to:

- Demonstrate that the POI algorithm was capable of inducing knowledge-based planning operators.
- Investigate empirically the within- and across-domain complexity of the POI algorithm.

The single-agent DUC-ASS program was used; this program was described in Chapter 3. Unless otherwise stated, the Single-Actor/Single-State-Change (SA/SSC) meta-heuristic[41] was employed.

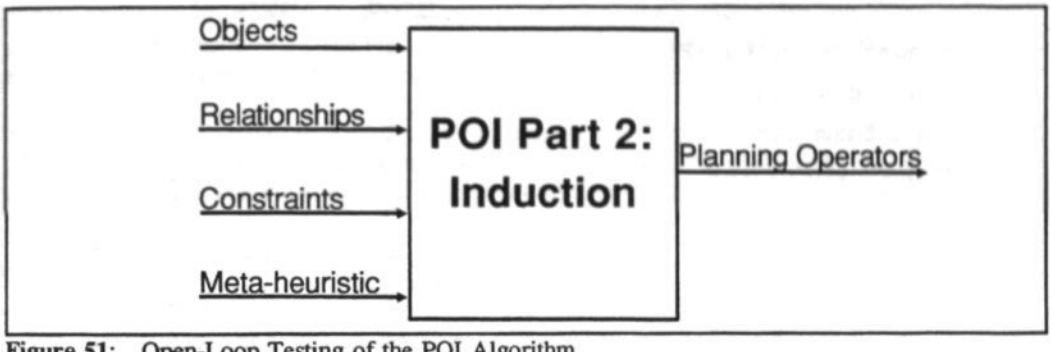

Figure 51: Open-Loop Testing of the POI Algorithm.

The experimental results are too voluminous to document in full. Instead, this chapter outlines the overall shape of the experimental programme, with notable results being highlighted. Section 4.1 summarises the domains for which DUC-ASS runs have been done. Section 4.2 details the design of the experimental programme, concentrating on three of the domains. Section 4.3 demonstrates that the POI algorithm is capable of inducing knowledge-based planning operators for a well-known AI domain: the blocks world. Section 4.4 presents the results of the within-domain complexity investigations, and Section 4.5 presents the results of the across-domain complexity investigations. The rest of the chapter highlights the key results from:

- varying domain granularity (Section 4.6).
- varying domain constraints (Section 4.7).
- varying the meta-heuristic (Section 4.8).
- partitioning the version-space (Section 4.9).

---

[41] This meta-heuristic was described in Section 1.2.2.

- introducing inheritance in the domain model (Section 4.10).
- investigating the POI meta-domain (Section 4.11).

Section 4.12 draws conclusions.

4.1 DOMAINS

The domains for which DUC-ASS runs have been done are listed in Figure 52, together with the number of object-classes, primary relationship-classes, and USEABLE constraints in the domain. The number of all possible binary constraints is shown in brackets. Each domain is described briefly in the sub-sections below. The three domains on which the experimental programme concentrated - Piano-Playing, Blocks World, and High Performance Capillary Electrophoresis - are marked with asterisks.

	Domain	Object-classes	Relation-classes	Constraints
	Finger-crossing (F)	1	1	8 (11)
*	Piano-playing (P)	2	1	4 (4)
	Tank-farm (T)	2	2	11 (17)
*	Blocks World (B)	3	3	18 (39)
	Dining philosophers (D)	5	5	26 (44)
	Aircraft scheduling (A)	9	10	57 (120)
	Planning Operator Induction	19	25	202 (370)
*	High Performance Capillary Electrophoresis (H)	24	18	116 (181)

Figure 52: Experimental Domains.

4.1.1 Finger-Crossing Domain

The Finger-Crossing domain models the physical process of crossing two or more fingers. There is a single class of objects - `Finger` - which can be related by the single `crossing` relationship-class (see Figure 53). Several variants of the Finger-Crossing domain are possible, depending on the cardinality constraints on the `crossing` relationship-class. A one-to-one constraint is the baseline.

The Finger-Crossing domain is chiefly important in that it is the simplest possible domain that can be represented using the POI ontology. There are no meaningful domains with fewer object- or relationship-classes. Although it has a cardinality constraint, the Finger-Crossing domain is too small to exhibit exclusion constraints. Despite its extreme simplicity, the Finger-Crossing domain can be claimed to model a real-world domain. Opie and Opie (1959) recorded that finger-crossing is of ritual significance amongst school children in swearing oaths, preventing bad luck, and calling truces.

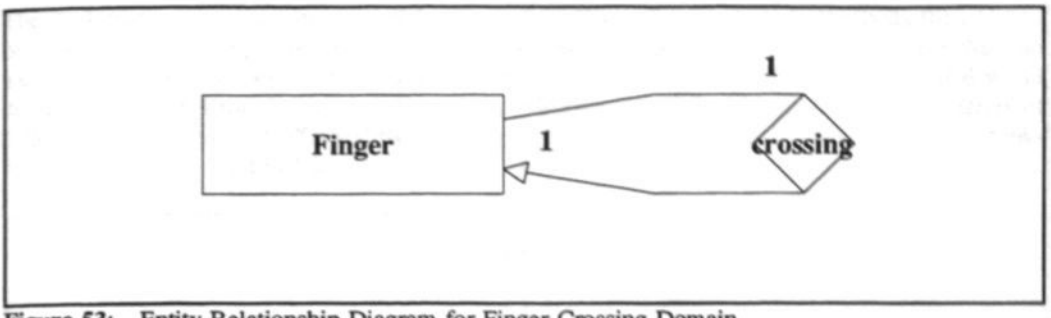

Figure 53: Entity-Relationship Diagram for Finger-Crossing Domain.

4.1.2 Piano-Playing Domain

The Piano-Playing domain models the physical process underlying the use of keyboard-based devices in making music (e.g., pianos, organs, harpsichords, harmonicas, and the like) and in interacting with symbolic systems (e.g., typewriters, computers, process-control systems, and the like). There are two classes of object: `Finger` and `Key`. Instances of these object-classes can related by the single `pressing` relationship-class (see Figure 54). Although various cardinality constraints on `pressing` are possible, the baseline for the experimental programme was one-to-one.

The Piano-Playing domain is a minimal elaboration of the Finger-Crossing domain, involving the addition of a second object-class. There is still only one relationship-class. Like Finger-Crossing, it is possible to claim that Piano-Playing is a real-world domain. The importance of the Piano-Playing domain rests on two aspects:

- *Action in the Piano-Playing domain can be plan-driven.* Pianists press and release keys in a partial order determined by a set of marks printed on sheets of paper. Such sheet music can be regarded as a plan which is generated by the composer and executed by the pianist. Improvisation corresponds to reactive planning. Similarly, a secretary who types a letter from dictation notes can be seen as executing the plan generated by the person who dictated the letter.

- *Without constraints, the Piano-Playing domain exhibits a massive combinatorial explosion.* As pianos usually have 88 keys and pianists have ten fingers, the unconstrained version space would have 2^{880} nodes. This is substantially more than the number of atoms in the Universe. In real life, piano-playing is possible because it is highly constrained. The cardinality constraints restrict each finger to pressing just one key at a time, and each key to being pressed by no more than one finger at a time. These cardinality constraints are modelled in the Piano-Playing domain. In addition, fingers are linearly ordered, grouped in fives by being affixed to a hand, with each hand having a limited span of about 15 keys. The keys are also linearly ordered. These additional real-world constraints are not modelled in the Piano-Playing domain because they would require additional object- or relationship-classes[42].

[42] There is a branch of experimental music concerned with composing *impossible music*, i.e., music that cannot be played by humans. Impossible music can be modelled by relaxing one or more of the constraints mentioned.

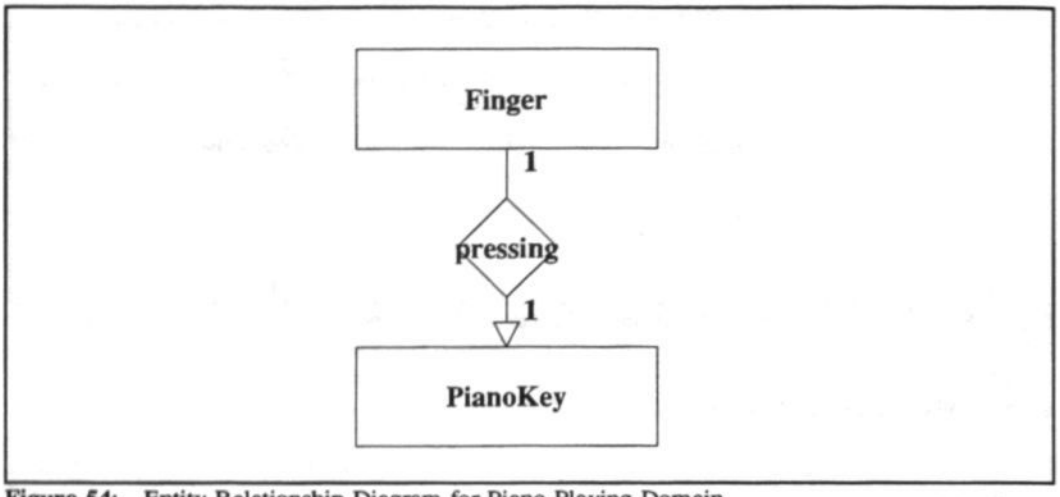

Figure 54: Entity-Relationship Diagram for Piano-Playing Domain.

4.1.3 Tank-Farm Domain

The Tank-Farm domain models the processes common to oil refineries and chemical plants, which consist of a network of vessels, joined by pipes through which fluids flow. Such real-world domains are often collectively known as "tank farms". Two classes of vessel are distinguished (see Figure 55): the `Tank` object-class models storage vessels, and the `Reactor` object-class models vessels in which fluids react with one another. Pipes and fluids are not modelled explicitly. When fluid flows from a `Tank` to a `Reactor`, this is denoted by saying that the `Tank` is `feeding` the `Reactor`. Similarly, fluid flow from a `Reactor` to a `Tank` is modelled as the existence of a `loading` relationship between them. As baseline, it is assumed that `feeding` and `loading` both have one-to-one cardinality constraints. Additional exclusion constraints include the assumptions that a `Tank` cannot both be feeding and being loaded simultaneously, and that a `Reactor` cannot both be loading and being fed simultaneously.

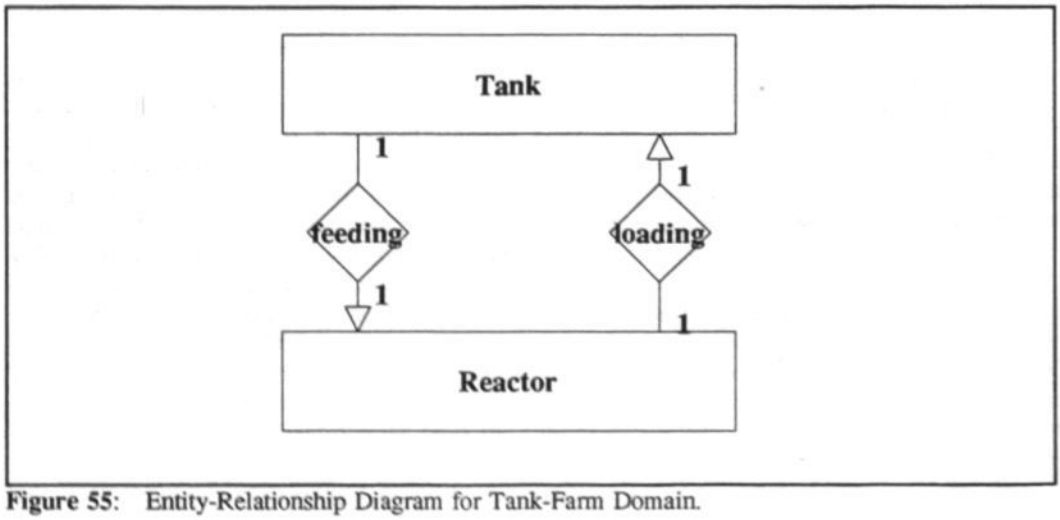

Figure 55: Entity-Relationship Diagram for Tank-Farm Domain.

The Tank-Farm domain is a minimal elaboration of the Piano-Playing domain, involving the addition of a second relationship-class. Having two relationship-classes, it is the smallest domain that can exhibit exclusion constraints. Once again, it is possible to claim that the Tank-Farm is a real-world domain. The Tank-Farm domain is important in that there is a close analogy between the effects of different meta-heuristics in POI and the effects of different safety policies in operating real-life tank-farms, as shown in Section 4.8.

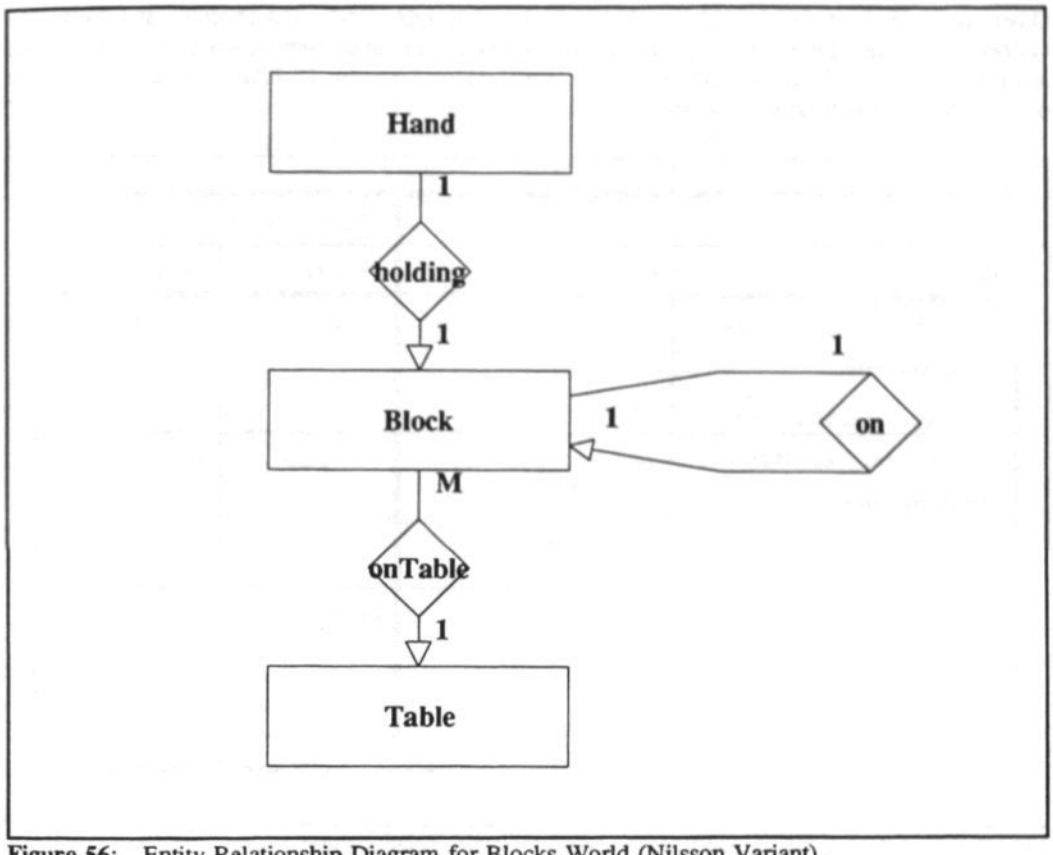

Figure 56: Entity-Relationship Diagram for Blocks World (Nilsson Variant).

4.1.4 Blocks World

The Blocks World models the processes found in using a (robot) hand to manipulate a set of children's blocks stacked on a table. The origins of the Blocks World can be traced back to Winograd (1972), but the baseline in my experimental programme is modelled on Nilsson (1980, pps. 275-283). This baseline will be known as the "Nilsson variant". Nilsson provided neither a domain description nor any domain constraints. A Nilsson-compatible domain description may be found in Rich and Knight (1991, pps. 332-333), together with five domain constraints. Another Blocks World representation compatible with the Nilsson variant may be found in Thornton and du Boulay (1992, pp. 171-200).

An entity-relationship model for the Nilsson (1980) variant of the Blocks World has already been shown in Chapter 2. For convenience, it is reproduced here again at Figure 56. The POI ontology requires the systematic identification of primary, converse, and inverse relationships, not all of which

are to be found in Nilsson (1980) (or in other blocks-world references). Figure 57 compares the baseline model used in the POI experiments with Nilsson's representation. Three classes of objects are distinguished: `Block`, `Hand`, and `Table`. The `Block` object-class represents the children's blocks that are stacked. The `Hand` object-class represents the robot hands that are capable of manipulating the blocks. The `Table` object-class represents those objects which support the stacks of blocks being manipulated. Three primary relationship-classes are distinguished. `Hands` and `Blocks` are related by the `holding` relationship, `Blocks` and `Tables` by `onTable`, and one `Block` and another by `on`. Other predicates identified by Nilsson (1980), Rich and Knight (1991), and Thornton and du Boulay (1992) are inverses of these primary relationship-classes. For example, Nilsson's HANDEMPTY is the actor-inverse of `holding`, which I denote as `notHolding`. Likewise, CLEAR(x) is the actee-inverse of `on`, which I denote as `notBeneath`.

Baseline model		Nilsson's (1980) representation	
Primary	Inverse	Primary	Inverse
`[holding ?Hand1 ?Block1]`	`[notHolding ?Hand1]`	`HOLDING(x)`	`HANDEMPTY`
and its converse:			
`[heldBy ?Block1 ?Hand1]`	`[notHeld ?Block1]`	-	-
`[on ?Block1 ?Block2]`	`[notOn ?block1]`	`ON(x, y)`	-
and its converse:			
`[beneath ?Block2 ?Block1]`	`[notBeneath ?Block2]`	-	`CLEAR(x)`
`[onTable ?Block1 ?Table1]`	`[notOnTable ?Block1]`	`ONTABLE(x)`	-
and its converse:			
`[supporting ?Table1 ?Block1]`	`[notSupporting ?Table1]`	-	-

Figure 57: Comparison of Baseline and Nilsson (1980) Relationship-Classes.

The cardinality constraints in Figure 56 follow the conventions that:

- A `Hand` can only hold one `Block` at a time.
- A `Block` can only be held by one `Hand` at a time.
- A `Block` can only be on top of one other `Block` at a time.
- A `Block` can only be beneath one other `Block` at a time.
- A `Block` can only be on one `Table` at a time.
- A `Table` can support many `Blocks` at a time.

Beyond the six exclusion constraints between the primary/converse relationships and their inverses,

there are additional exclusion constraints, including:

- A `Block` cannot be both held by a `Hand` and on a `Table`.
- A `Block` cannot be both on another `Block` and on a `Table`.
- A `Block` cannot be both held by a `Hand` and beneath another `Block`.
- A `Block` must either be held by a `Hand`, or on another `Block`, or supported by a `Table`.

These constraints are only partially documented in the literature. For example, Rich and Knight (1991) list two of the cardinality constraints, two of the exclusion constraints between primaries and their inverses, and one of the additional exclusion constraints. Drummond (1990) and Tenenberg (1990) have each documented three of the block world constraints.

Reference	Inputs	Operators
Nilsson, 1980, pp. 275-283.	Domain description, state-transition network, drawings of two states.	4
Bundy, Burstall, Weir and Young, 1980, pp. 51-52.	Domain description, drawings of three states.	1
Charniak and McDermott, 1985, pp. 490, 515-517, Ex 9.34.	Domain description, drawing of one state.	3 (1 left as an "exercise for the reader")
Ramsay and Barratt, 1987, pp. 13-16, 49-51.	Domain description, list of object-classes, drawings of two states.	8
Genesereth and Nilsson, 1987, pp. 263-281 (Ch 11).	Text, drawings of two states, state-space, state-transition network, and example procedure.	3
Rich and Knight, 1991, pp. 332-336.	Domain description, lists of relations and some constraints, drawing of one state	4
Thornton and du Boulay, 1992, pp. 171-200 (Ch 7).	Detailed description, drawing of one state, several example runs of POP-11 and Prolog code.	4

Figure 58: AI Textbook References to Blocks World.

The baseline Blocks World can be regarded as the composition of the Piano-Playing and Finger-Crossing domains. The Piano-Playing domain can be recognised in two places: firstly in the grouping of the `Hand` and `Block` object-classes with the `holding` relationship, and secondly in the grouping of the `Block` and `Table` object-classes with the `onTable` relationship-class. The Finger-Crossing domain can be recognised in the grouping of the `Block` object-class with the `on` relationship-class.

It is possible to claim that the Blocks World is a real-world domain. This claim would be challenged by some AI researchers on the grounds that it is a toy domain. In justification, one may point to the analogy between the Blocks World and a container terminal, as found in major harbours such as Rotterdam. Stacks of containers (cf. `Blocks`) are manipulated by transporters (cf. `Hands`) on parking areas or in ships' holds (cf. `Tables`). Companies such as European Container Terminals use standard procedures containing sequences of container-manipulation actions (cf. blocks-world plans and operators).

The Blocks World is important in that it is often used as an example in the knowledge-based planning literature. Figure 58 lists in chronological order the main AI textbook references to STRIPS-style Blocks World operators.

Analysis (Grant, van den Herik and Hudson, 1994) showed that the references fall into three groups, as follows:

- *The three-object-class group*. The members of the three-object-class group are Nilsson (1980), Rich and Knight (1991), and Thornton and du Boulay (1992). They model the blocks world using three object-classes and three primary relationship-classes, and they identify four operators. The *Nilsson variant* will be taken as the representative of this group.

- *The two-object-class group*. The members of the two-object-class group are Bundy, Burstall, Weir and Young (1980), Charniak and McDermott (1985), and Genesereth and Nilsson (1987). They model the blocks world using two object-classes and two primary relationship-classes, and they (should) identify three operators[43]. The *Genesereth and Nilsson variant* will be taken as the representative of this group.

- *The four-object-class group*. Ramsay and Barratt (1987) - the single member of this group - model the blocks world using four object-classes and four primary relationship-classes, identifying eight planning operators.

Detailed comparison between Nilsson (1980) and Genesereth and Nilsson (1987) brings further insights. Despite having an author in common (Nilsson), their variants differ by one object-class. The Genesereth and Nilsson variant omits the `Hand` object-class, as shown in Figure 59. Both references depict the complete state-space for the three-blocks world. In Nilsson (1980, Figure 7.3, p. 283) there are 22 states, and in Genesereth and Nilsson (1987, Figure 11.3, p. 265) there are only 13. Inspection shows that, in the "missing" nine states, the hand is holding a block. Since the `Hand` object-class was omitted by Genesereth and Nilsson, they were unable to represent the "missing" states.

Nilsson (1980) and Genesereth and Nilsson (1987) also identify different operator-sets. In the Nilsson variant, picking up a block and stacking it on another are two separate operations. By contrast, this is a single operation in the Genesereth and Nilsson variant, becoming their **`stack`** (`S`) operator. Similarly, Genesereth and Nilsson's **`move`** (`M`) operator unites the Nilsson-variant operations of unstacking a block from one stack and stacking it onto another.

---

[43] I write "should" because, strictly speaking, not all group members have done so. Bundy, Burstall, Weir and Young (1980) quote just one MOVE operator, but add [CLEAR FLOOR] to the add-list "because it is inadvertently deleted in case (c)" (*ibid.*, p. 51). They failed to see that this is a sign that different MOVE operators are needed according to the object-class of the starting and finishing supports. Although Charniak and McDermott (1985, pps. 515-6) do not make the same mistake, they do quote two **move** operators and then inform the reader that "Exercise 9.34 asks you to write a third schema defining **move**".

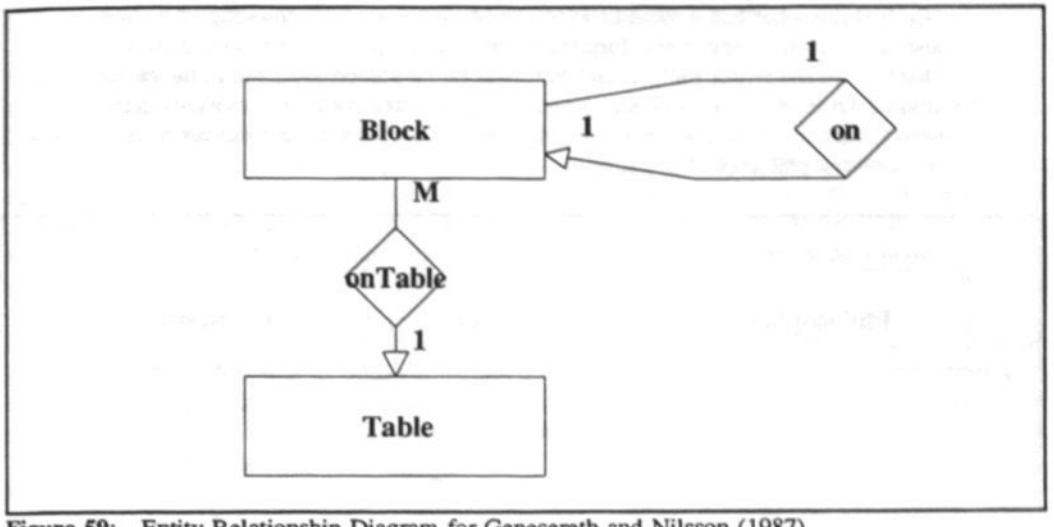

Figure 59: Entity-Relationship Diagram for Genesereth and Nilsson (1987).

The existence of multiple variants of the Blocks Worlds opens up the possibility of experimentation with the help of the POI algorithm to reproduce the multiplicity, to explain why this multiplicity arises, and to relate the variants to one another. Sections 4.6 and 4.7 report the results of such experiments.

Moreover, on the evidence of multiple variants of the Blocks World, the following hypotheses can be proposed:

- For a given domain, an increase in the number of object-classes is associated with an increase in the number of relationship-classes. In the Blocks World, the association is linear, at one relationship-class per object-class[44].

- For a given domain, an increase in the number of relationship-classes is associated with an increase in the number of planning operators.

These hypotheses would have to be verified by comparing variants for other domains. Verification must await the development of another domain with multiple variants.

A key conclusion can be drawn without awaiting confirmation from other domains. In writing the textbooks, each author listed in Figure 58 judged that he had designed a satisfactory model of the Blocks World. Nevertheless, I have shown that the models differ. Therefore, I conclude that the number of object-classes needed to represent a given domain is - to some extent - a matter for design judgement.

4.1.5 Dining Philosophers Domain

The Dining Philosophers domain models the processes found in an example domain proposed by Dijkstra (1971). As described by Hoare (1985, p. 75):

[44] A possible reason for the ratio of one relationship-class per object-class is discussed in Section 4.10.1.

"Each philosopher had a room in which he could engage in ... thinking; there was also a common dining room, furnished with a circular table, surrounded by five chairs ... To the left of each [chair] there was laid a golden fork, and in the centre stood a large bowl of spaghetti ... Such is the tangled nature of spaghetti that a second fork is required to carry it to the mouth. ... Of course, a fork can be used by only one philosopher at a time."

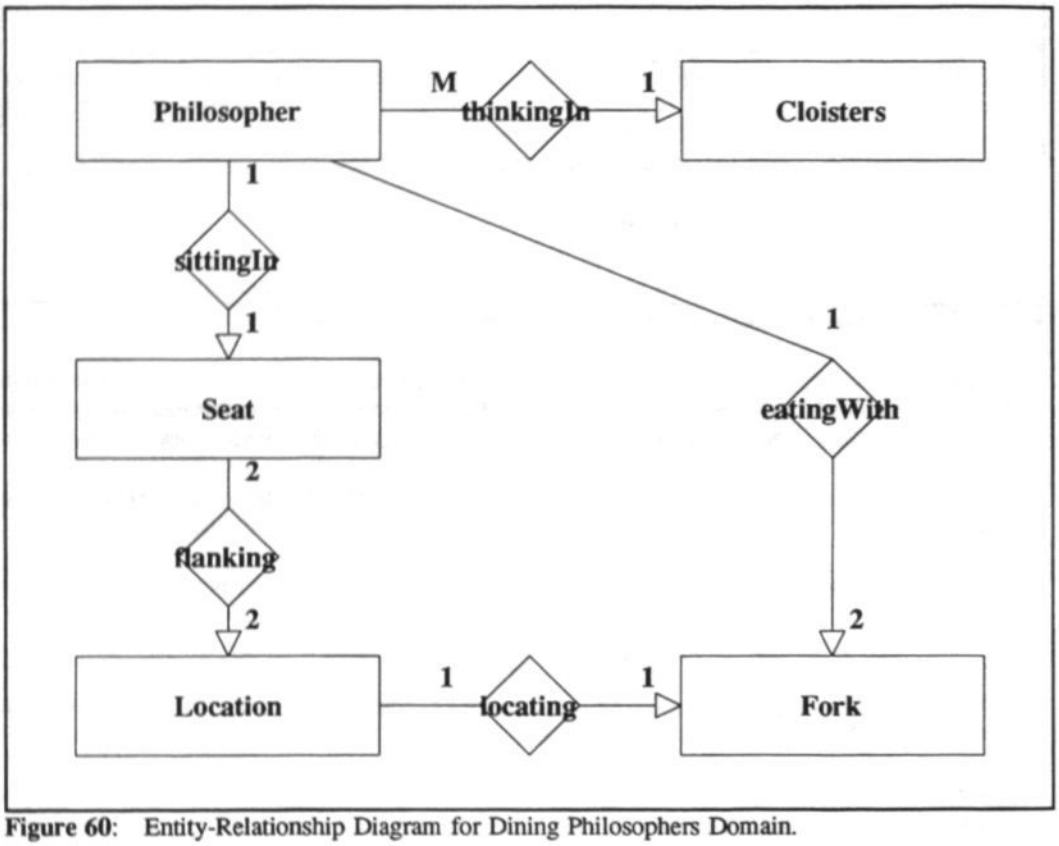

Figure 60: Entity-Relationship Diagram for Dining Philosophers Domain.

I distinguish five object-classes in the Dining Philosophers domain (see Figure 60): `Philosopher`, `Cloisters`, `Fork`, `Location`, and `Seat`. The `Philosopher` object-class represents the philosophers who either think or (try to) eat sitting at the circular table. The `Cloisters` object-class represents the rooms where the `Philosophers` do their thinking. The `Fork` object-class represents the cutlery items used by the `Philosophers` when eating. The `Location` object-class represents the places on the table where the `Forks` rest when not in use. The `Seat` object-class represents the seats which the `Philosophers` must sit in when eating. By convention:

- There is just one instance of the `Cloisters` object-class.
- There are equal numbers of `Philosophers`, `Seats`, `Forks`, and `Locations`. The usual number is five instances of each object-class.
- The `Seats` are equally placed around the table, with a `Location` between each pair of `Seats` and a `Seat` between each pair of `Locations`.

- A Philosopher must have two Forks before he/she can eat.

Five relationship-classes are distinguished. Philosophers may be thinking in the Cloisters, eating with (a pair of) Forks, and sitting in Seats. The Seats are flanking (a pair of) Locations, which may be locating Forks. Cardinality constraints ensure (for example) that a Philosopher can only be sitting in one Seat at a time, that a Seat can only hold one Philosopher at a time, and that a Location can only locate one Fork at a time. Additional exclusion constraints include that a Philosopher cannot be both thinking (in the Cloisters) and eating (with Forks) at the same time, and that a Philosopher cannot be eating without sitting (in a Seat).

It should be noted that my model of the Dining Philosophers domain would need to be refined, if the aim of experimentation was to obtain a perfect domain model. For example, it is not possible to represent as a binary constraint the requirement for a Philosopher to have two Forks before eating. Since my aim is merely to investigate the complexity of the POI algorithm, it is unnecessary to eliminate the remaining imperfections. Section 4.10 shows how similar imperfections in the Blocks World can be compensated for.

Dijkstra's (1971) "dining philosophers' problem" has had great influence on the theory of concurrent programming. It is a benchmark of the expressive power of concurrent programming languages, because it exhibits the classic problems of mutual exclusion, deadlock, and lockout. It has been analysed in a number of different ways, e.g., by Hoare (1985), Kokol (1987), and Ringwood (1988). As for the blocks world, there are a number of variants. Differences range from the superficial (e.g., rice and chopsticks instead of spaghetti and forks) to the vital (e.g., each chair in Hoare's account is reserved for a particular philosopher, as in a London club).

4.1.6 Aircraft Scheduling Domain

The Aircraft Scheduling domain models the processes found in airliner maintenance. I distinguish nine object-classes in the Aircraft Scheduling domain (see Figure 61). The Flight object-class represents a flight to be flown, i.e., an element of the airline's timetable. The Aircraft object-class represents the airliners which the airline operates. The Payload object-class represents the passengers or freight which the airliners must transport. The FuelTruck object-class represents the vehicles which fill the Aircraft with fuel for their next flight. The Part object-class represents those components of the Aircraft that can become faulty. The Fault object-class represents the failures which can be manifested by the Aircraft's Parts. The Tradesman object-class represents the airline's personnel who refuel, repair, and load the Aircraft. The Skill object-class represents the set of skills required of the Tradesmen in order to operate FuelTrucks, to repair Faults, and to load Payloads. The Tool object-class represents the tools and other equipment used in refuelling, repairing, or loading activities.

Ten relationship-classes are distinguished. Tradesmen may use Tools to operate FuelTrucks, to repair Faults, and to load Payloads. They need to exhibit the appropriate Skill for each activity. Aircraft consist of Parts. Parts can manifest Faults. Aircraft can be refuelled by FuelTrucks, can carry a Payload, and can fly Flights.

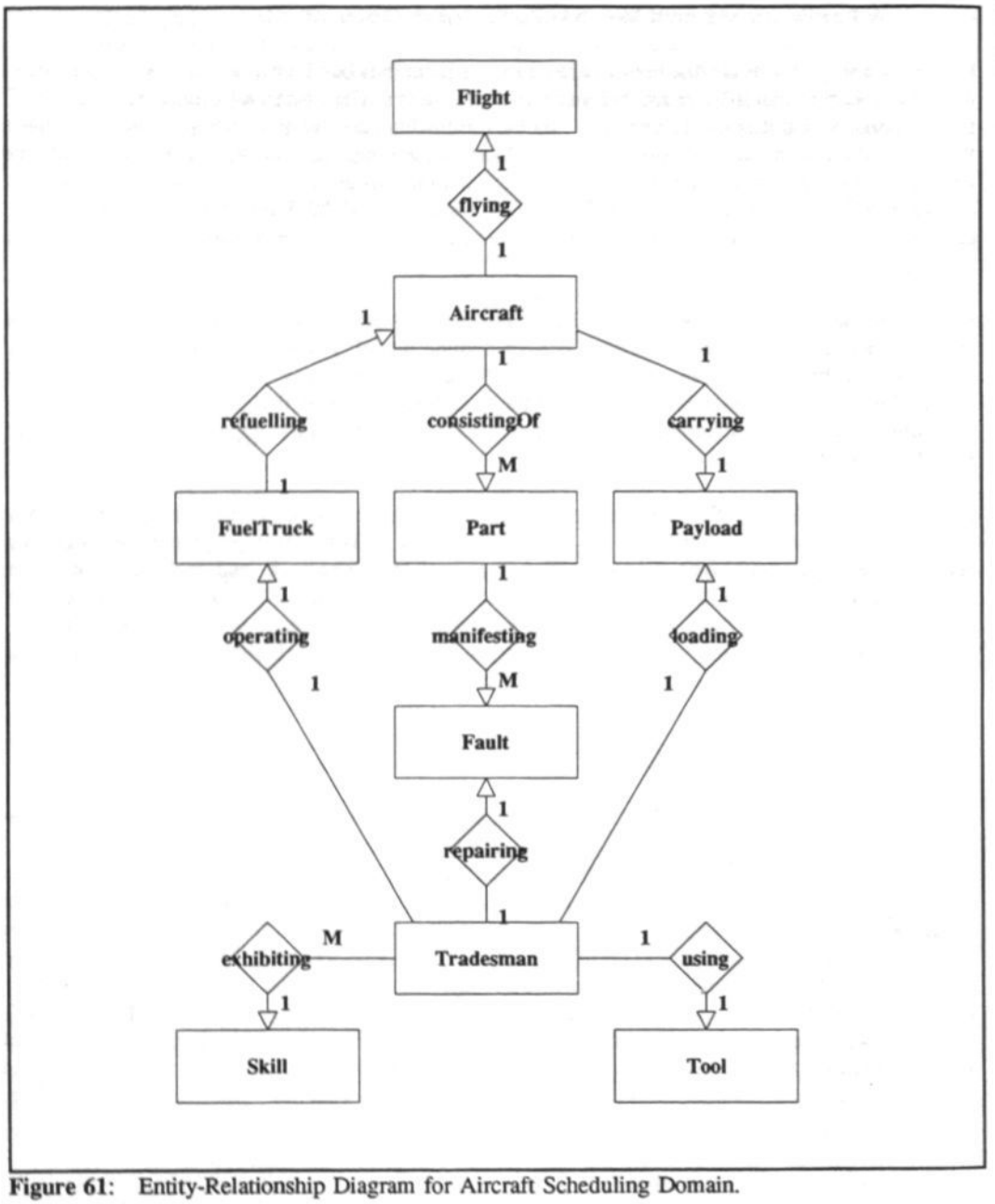

Figure 61: Entity-Relationship Diagram for Aircraft Scheduling Domain.

Cardinality constraints ensure (for example) that an `Aircraft` can carry only one `Payload` at a time and that a `Payload` can only be carried by one `Aircraft` at a time. Similarly, though a `Tradesman` can be using many `Tools`, a given `Tool` can only be used by one `Tradesman` at a time. Question-marks show cardinalities that are dependent on the type of `Aircraft` or on the airline's operating policy. For example, there are arrangements by which one large `FuelTruck` can refuel several small `Aircraft` simultaneously. On the other hand, several `FuelTrucks` may be needed at the same time to refuel a very large `Aircraft`. Airline policy will determine whether

Tradesmen exhibit one or many Skills (i.e., are "single-skilled" or "multi-skilled"). The baseline for this thesis is that refuelling is a one-to-one relationship and that exhibiting is many-to-one.

Additional exclusion constraints ensure that:

- A Tradesman can only be operating a FuelTruck, or repairing a Fault, or loading a Payload.
- A Tool must be used to do any of these things.
- Each Tradesman does only those activities for which he/she has the appropriate Skill(s).
- An Aircraft cannot be flying a Flight if it is being refuelled, has a Part which manifests a Fault, or is not carrying a Payload.

As for the Dining Philosophers domain, there are imperfections in my model of the Aircraft Scheduling domain. Refinement of the domain model is not necessary for the purposes of my research.

The Aircraft Scheduling domain is a real-world domain that is to be found at all major airports. As well as performing scheduling in real aircraft operations myself, I have studied the domain extensively using simulation techniques. I concluded (Grant, 1985) that aircraft schedulers must make decisions in state-spaces which potentially contain some 2^{540} states. The mean time between decisions is about 15 minutes. Fortunately for the schedulers' sanity, the problem is very highly constrained, with at most 2 or 3 of the 2^{540} possible decisions being valid at any decision-point.

4.1.7 High Performance Capillary Electrophoresis Domain

The High Performance Capillary Electrophoresis (HPCE) domain models the processes found in a particular class of laboratory instruments. HPCE is a general technique for analysing chemical mixtures (Eckhard, 1992). The technique is in routine use in laboratories world-wide. Several manufacturers supply HPCE instruments. The HPCE domain in this thesis is based on the P/ACE System 2000 instrument, supplied by Beckman Instruments Nederland b.v. The Beckman instrument has three parts (Beckman, 1989): an analyser, a Programmable Logic Controller (PLC), and a PC-based data analysis system. The HPCE domain models only the analyser.

An HPCE analyser is depicted schematically in Figure 62. The chemical mixtures to be analysed and other fluids (e.g., capillary-cleaning fluid ("regenerator solution"), de-ionised water, and electrically-neutral buffer solution) are placed in vials in one or more trays. To analyse a chemical mixture, the capillary is rinsed with cleaning fluid and de-ionised water, and then filled with the buffer solution. Since the long, thin capillary is fragile, it is held in a cartridge. The vial containing the chemical mixture is raised to the inlet receptacle at one end of the cartridge. Another vial is raised to the outlet receptacle. The action of raising a vial to a receptacle causes the vial to be sealed off for pressurisation and an electrode to be dipped into the fluid contained in the vial. The pressure supply is briefly switched on, to introduce a minute quantity of the mixture into the capillary. After the pressure supply has been switched off, the power supply is switched on. This causes a high voltage to be applied across the capillary, from the vial at the inlet receptacle to the vial at the outlet receptacle. Under the influence of the high voltage, the contents of the capillary moves. The direction and speed of movement depends on the mixture's chemical composition and concentration. The effect is to split the mixture into its component chemicals. These chemical components pass under a detector, which

generates data. Typically, the capillary and its contents are illuminated by filtered ultra-violet light, with the light absorption against time being measured by the detector. The various parts of the HPCE analyser are designed as replaceable modules, which are assembled into the instrument's body prior to use.

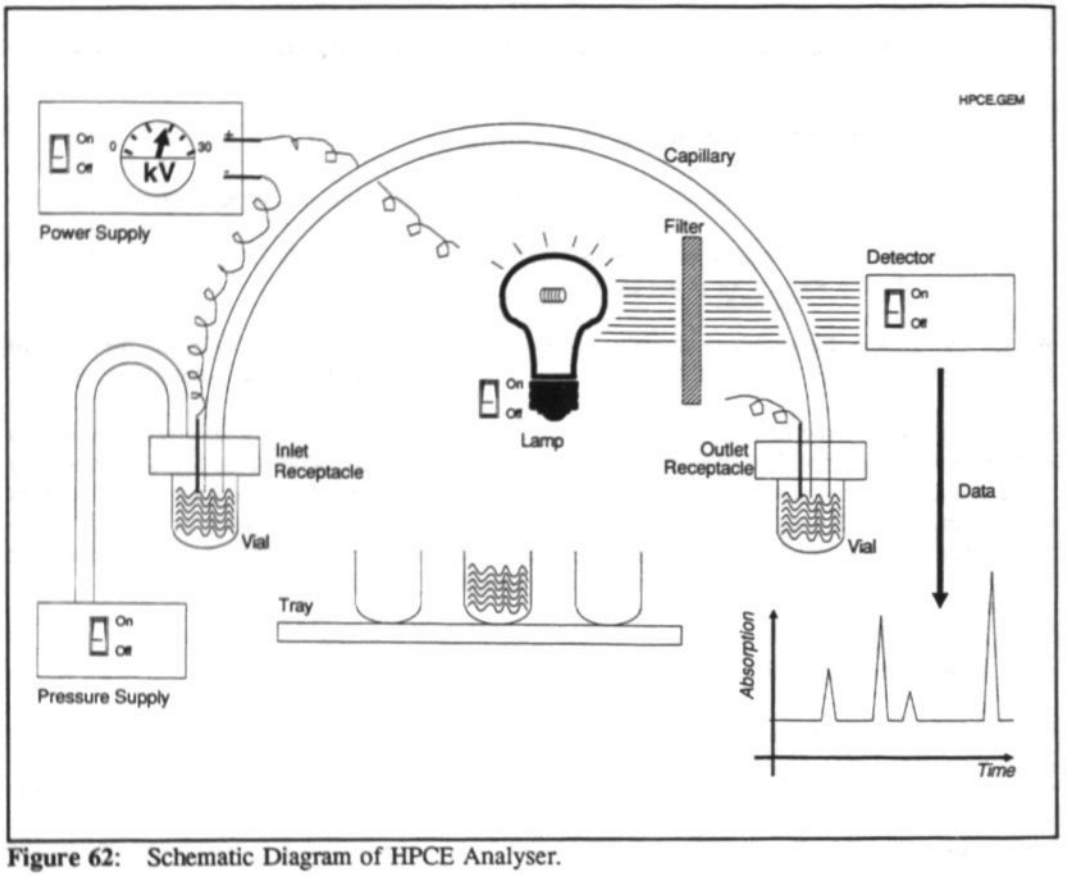

Figure 62: Schematic Diagram of HPCE Analyser.

I distinguish 24 object-classes in the HPCE domain (see Figure 63). The `InstrumentBody` object-class models the HPCE analyser's metal framework. The `Module` object-class is an abstract superclass which models the ability to fit and replace modules into the instrument body prior to use. The `Cartridge` object-class represents the cartridge that holds the `Capillary`. The `Side` object-class represents the two sides of the capillary, i.e., inlet and outlet. The `OnOffModule` object-class is an abstract specialisation of the `Module` object-class representing those (replaceable) modules that can be switched on and off. The `On` object-class represents the states that an `OnOffModule` can be in; there is just one instance of this class. Other object-classes which specialise the `Module` and `OnOffModule` superclasses represent the objects shown in Figure 62.

The `Container` object-class is an abstract superclass which models the ability to contain fluids. `Containers` can contain one or more `FluidUnits`, each consisting of a single `Chemical`. `Chemical` is an abstract superclass that is specialised into `Gas` and `Liquid` object-classes. The `Capillary` subclass specialises the `Container` superclass in that the `FluidUnits` it contains are ordered linearly along its length. By contrast, `FluidUnits` contained in `Vials` are randomly ordered. The `Light` abstract superclass is specialised into `FilteredLight` and `ModulatedLight` object-

classes.

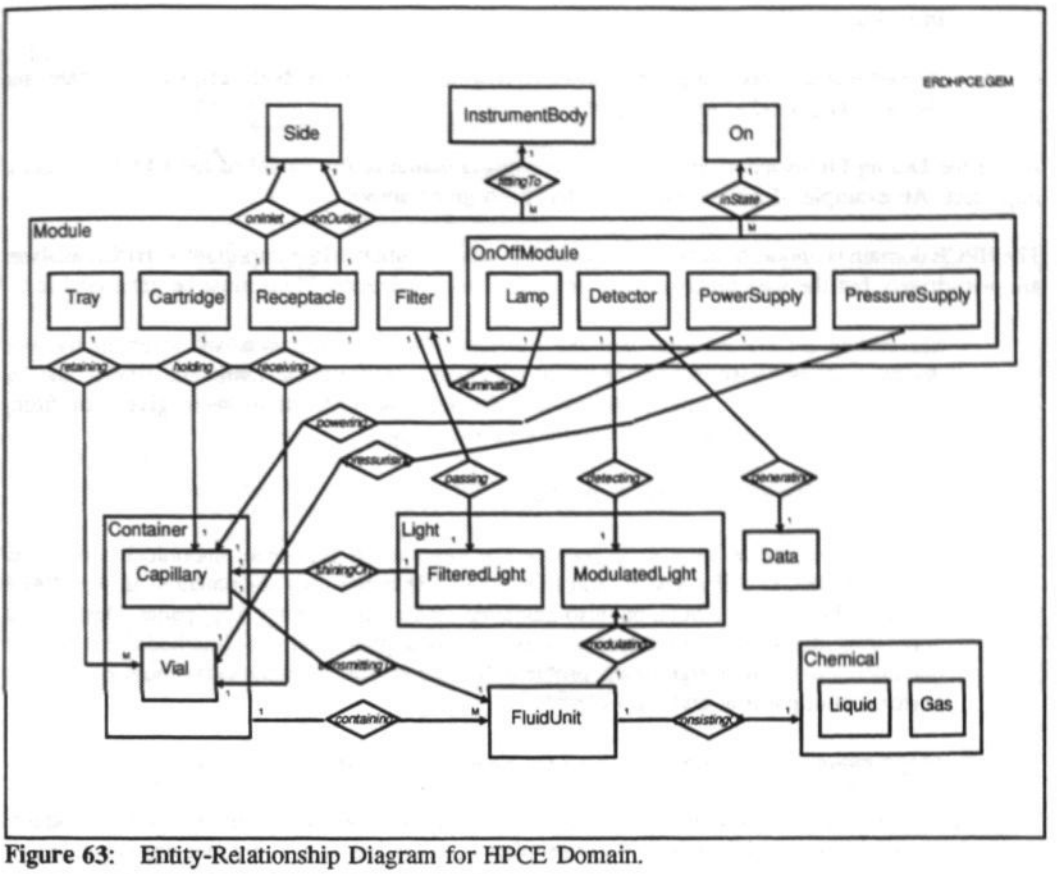

Figure 63: Entity-Relationship Diagram for HPCE Domain.

I distinguish 18 relationship-classes. `Modules` can be `fitting to` an `InstrumentBody`, and `OnOffModules` can also be `in the On state`. `Trays retain Vials`, which may be `received` by `Receptacles` and `pressurised by a PressureSupply`. A `Receptacle` may be either `on the inlet` or `on the outlet Side`. `Capillaries` may be `held by a Cartridge`, and `powered by a PowerSupply`. `Containers` may `contain FluidUnits`, each `consisting of a Chemical`. `Lamps illuminate Filters`, which `pass FilteredLight`. The `FilteredLight shines on` the `Capillary`, which `transmits` it to the `FluidUnits` it contains. These `FluidUnits modulate` the `ModulatedLight`, which is `detected by` the `Detector`. The `Detector` then `generates Data`.

Cardinality constraints ensure (for example) that a `Receptacle` can receive only one `Vial` at a time, that a `Lamp` can illuminate only one `Filter` at a time, that a `Tray` can hold many `Vials`, and that a `FluidUnit` can consists of only one `Chemical`. Additional exclusion constraints ensure (for example) that:

- A `Receptacle` can only be `on the inlet` or `on the outlet Side` at a time.
- A `Capillary` can only be `powered` when it is `held by a Cartridge`.

- A `Vial` can only be `pressurised` if it is `received by` a `Receptacle` `on the inlet Side`. Because this is a triple constraint, it cannot be represented in the DUC-ASS program.

- A `Detector` can only be `generating Data` if it is both `in` the `On` state and `detecting ModulatedLight`.

As for the Dining Philosophers and Aircraft Scheduling domains, my model of the HPCE domain is imperfect. An example of a triple constraint has been given above.

The HPCE domain is unquestionably a complex, real-world domain. More importantly, HPCE analyses are plan-driven, but the user must generate several plans in advance. Plans must be generated for:

- *assembling all the modules into the instrument-body.* Instructions are given in the user documentation for fitting the capillary cartridge, the trays, the lamp, the filters, and the detector, but not for other modules. In particular, no instructions were given for fitting receptacles or for embedding a capillary in a cartridge.

- *filling vials and placing them in the trays.*

- *programming the instrument to perform an analysis.* A high-level canonical plan (PRE-RINSE, INJECT, SEPARATE, and POST-RINSE) is not only documented in the P/ACE System 2000 users' manual, but also engraved on the instrument's front panel. Despite this, generating detailed instantiations of this canonical plan - termed "methods" in the user documentation - is a significant problem for users. The documentation advises users to perform multiple trial-and-error runs.

- *programming the instrument to perform a series of analyses.*

No planning operators are given in the user documentation. Therefore, any planning operators induced by the POI algorithm would be novel. Because other operator-learning techniques are dependent on the pre-existence of a plan, they would be confined to acquiring the planning operators in the canonical plan, i.e., **`rinse`**, **`inject`**, and **`separate`**. POI would demonstrate its advantage over other operator-learning techniques if it induced other operators and/or it decomposed **`rinse`**, **`inject`**, and **`separate`** to more primitive operators[45]. Figure 64 describes the decomposition of **`rinse`**, **`inject`**, and **`separate`**, as identified during the development of a software simulation of the HPCE instrument (Grant, Wigmans, Eckhart and van Eenennaam, 1992). This decomposition shows that the set of primitive operators would include ones modelling:

- placing a vial under a receptacle.

- raising a vial to the receptacle under which it had been placed.

- switching on an OnOffModule instance.

- waiting a period of time.

- switching off an OnOffModule instance.

---

[45] Decomposition would result in an abstraction hierarchy of operators, as in hierarchical planning (Sacerdoti, 1977).

- lowering a vial from the receptacle to which it had been raised.

RINSE

```
place vial containing rinse solution under inlet receptacle
raise vial to inlet receptacle
place empty vial under outlet receptacle
raise vial to output receptacle
switch on pressure supply
wait X seconds
switch off pressure supply
lower vial from inlet receptacle
lower vial from outlet receptacle
```

INJECT

Same as RINSE, but with:
- vial containing sample at inlet receptacle, rather than rinse solution, and
- small value for X (typically 5 seconds).

SEPARATE

```
place vial containing buffer solution under inlet receptacle
raise vial to inlet receptacle
place vial containing buffer solution under outlet receptacle
raise vial to outlet receptacle
switch on detector
switch on power supply
wait Y seconds
switch off power supply
switch off detector
lower vial from inlet receptacle
lower vial from outlet receptacle
```

Figure 64: Primitive Operators for HPCE.

4.2 METHODOLOGY

4.2.1 Basis for Experiment Design

There are two ways in which open-loop testing of the POI algorithm can be performed: against internal or external standards. The experimental programme included both.

4.2.2 Testing against Internal Standards

Testing against internal standards is designed to check that the values of the input and output variables remain within their valid ranges. This is done by *sensitivity analysis*, i.e., each of the inputs is varied and the resulting outputs are examined. The following input data-structures can be varied:

- The set of *object-classes*.
- The number of *instances* of each object-class.
- The set of *relationship-classes* connecting pairs of object-classes.

- The set of *constraints* connecting pairs of relationship-classes.

- The *meta-heuristic* used to identify candidate transitions.

A three-step procedure is applied for sensitivity analysis:

(1) With all inputs set to their *baseline* values, check that the outputs are within their expected ranges.

(2) For as many combinations as possible, vary each input around its baseline value, and check the validity of the outputs obtained (at least qualitatively).

(3) Examine the change in output resulting from a given change in input. Confirm that the output changes in the expected manner[46].

4.2.3 Testing against External Standards

Testing against an external standard is designed to establish that the values obtained from the POI algorithm match the corresponding values obtained from the external standard. There are two types of external standard:

- *A domain expert.* As well as being used as the source of knowledge in knowledge-based systems, experts can be used to assess automatically-generated knowledge. Using experts in this way will be termed *face validation.* The standards used are vague, but they exist as the expert's subjective impressions of the reality of the operators induced by the POI algorithm. A two-step procedure is applied:

  (1) Run the POI algorithm.

  (2) Obtain a domain expert's judgement on the realism of the planning operators it induces, together with the intermediate outputs such as the induced states and transitions.

- *An oracle.* Howden (1978) defines an *oracle* as any program, process, or body of data that predicts the expected outcome of a set of tests. In AI, oracles are used in machine learning as teachers and as sources of perfect knowledge, e.g., for rule induction on chess endgame databases (Dekker, van den Herik and Herschberg, 1990). In testing the POI algorithm, the outcomes will be variable values (i.e., in *variable-parameter validation*), events (i.e., in *event validation*), and relationships (i.e., in *hypothesis validation*). Application of event and hypothesis validation in testing the POI algorithm is deferred to Chapter 5 because these forms of testing are more appropriate to closed-loop experimentation. Two sets of variables and parameters can be used in variable-parameter validation:

  - *The output variables and parameters.* Comparison of the outputs of the POI algorithm against the outputs of the oracle will be termed *Output Comparison Testing.* A two-step procedure is applied:

---

[46] Since metric dimensions are outside the scope of this thesis, it is not possible to check the sign or the magnitude of changes in the outputs.

(1) Run both the POI algorithm and the oracle, with the same inputs.

(2) Compare their outputs.

- *The input variables and parameters.* Comparison of the inputs of the POI algorithm against the inputs of the oracle will be termed *Input Comparison Testing*. A three-step procedure is applied:

  (1) Run the oracle for a particular set of inputs, and note its output.

  (2) Run the POI algorithm, varying its input until it reproduces the output noted in (1).

  (3) Compare their inputs.

The use of an oracle is preferred because it is objective. Resort should be made to face validation by a domain expert only where no oracle is available.

4.2.4 Availability of Oracles

In principle, oracles must be found for each of the three illustrative domains. For open-loop testing, I sought oracles which:

- took as their input a list of domain objects, relationships and constraints,
- provided a set of knowledge-based planning operators as their output, and
- were trusted by domain experts or by the AI community. An indication of trustworthiness would be that the planning operators provided by the oracle had successfully undergone peer review.

Considering each domain in turn, I found that:

- *no oracle was available for the Piano-Playing domain.* This domain has not appeared before in the AI literature. However, it is such a simple domain that a set of operators can be readily constructed.
- *several possible Blocks World oracles were readily available.* I selected for analysis seven AI textbook references which gave Blocks World operator-sets in the STRIPS formalism. As already described, the Nilsson (1980) variant of the Blocks World has been selected as the baseline oracle. Other variants are compared in Sections 4.6 and 4.7.
- *only an expert was available for the HPCE domain.* Although a high-level canonical plan exists, no planning operators have been documented for HPCE instruments. Hence, no oracle is available. Resort to face validation by a domain expert was unavoidable.

4.2.5 Application to Illustrative Domains: Internal Standards

Figure 65 tabulates the experiments done using the three illustrative domains for sensitivity analysis, i.e., for comparison with internal standards. There is one row for each domain, and one column for each form of sensitivity analysis. For the Piano-Playing domain and the Blocks World, an experiment has been done for each intersection of a row with a column. For the HPCE domain, only the baseline experiment was done. In general, each experiment required many DUC-ASS runs.

The experiment names are shown in the table. The initial letters in the experiment names indicate the domain, i.e., "P" for the Piano-Playing domain, "B" for the Blocks World, and "H" for the HPCE domain. The numbers indicate the input parameter whose sensitivity is being analysed, e.g., experiments numbered "3" are those in which the number of object-instances is varied. It should be noted that variation of the object-classes or the relationship-classes (i.e., the experiments numbered "2" or "4") generally result in a change in the essential character of the domain.

Domains	Baseline	Internal standard (sensitivity analysis)				
		Object classes	Object instances	Relationship classes	Constraints	Meta-Heuristic
Piano-playing	P1	P2	P3	P4	P5	P6
Blocks world	B1	B2	B3	B4	B5	B6
HPCE	H1	-	-	-	-	-

Figure 65: Using Domains for Sensitivity Analysis.

In principle, all domains should be subjected to sensitivity analysis. In practice, some domains are more suited than others to variation in specific input variables. The following experimental variations were investigated:

- *Object-classes*. Varying the number of object-classes can have more or less dramatic effects, depending on the total number of object-classes in the domain. At one extreme, the Piano-Playing domain has so few object-classes that an addition or a reduction by one results in a different domain. For this reason, varying the number of object-classes is one aspect of across-domain complexity; more details are given in Section 4.5. At the other extreme, varying the number of object-classes in the HPCE domain by one alters its granularity but not its essential character. The Blocks World is the domain most suited to varying the number of object-classes, because the AI literature documents variants which differ in numbers of object-classes. More details are given in Section 4.6.

- *Object-instances*. It is always possible in any domain to vary the number of instances of each object-class. In general, varying the number of object-instances does not change the essential nature of the domain, but it affects substantially the induction runtime and memory requirements. For example, the same operator-set is induced for a three-blocks world as for a two-blocks world when the Nilsson (1980) representation is used, but the maximum possible size of the version-space lattice increases from 2^{16} to 2^{26} nodes. There are exceptions to the general rule for low and high numbers of object-instances. For example, the **stack** and **unstack** operators are not induced in a one-block world because at least two blocks

are needed for the on relationship. At high numbers of object-instances, the version-space becomes so large that its construction requires more memory than is available and/or the runtime becomes impracticably long. In domains with many object-classes the number of runs needed to vary the number of object-instances of each object-class also becomes impractical. For these reasons, the Piano-Playing domain was used for the within-domain complexity analysis reported in Section 4.4. Within the limits of available memory and runtime, similar analyses have been done for the Tank-Farm domain and Blocks World, but add little to the analysis done using the Piano-Playing domain. Furthermore, two countermeasures to the combinatorial explosion in the version-space were investigated. Section 4.9 reports on experiments into partitioning the version space, and Section 4.10 reports on the effects of introducing inheritance in the domain's object-classes.

- *Relationship-classes.* Varying the number of relationship-classes changes the essential nature of the domain. For example, the addition of one relationship-class to the Piano-Playing domain yields the Tank-Farm domain. Therefore, varying the number of relationship-classes is the second aspect of the across-domain complexity analysis reported in Section 4.5. The connectivity of the relationship-classes is also important. However, it is not possible to arrange connectivity along a dimension, as for varying numbers of relationship-classes. The importance of connectivity can only be illustrated by examples. Section 4.6.3 compares the Tank-Farm domain with the Genesereth and Nilsson (1987) variant of the blocks world. Although both domains have two object-classes and two relationship-classes, their differing connectivity results in the induction of different operator-sets.

- *Constraints.* There is no latitude for varying the total number of constraints in a domain, because this is directly dependent on the object-classes, relationship-classes, and their connectivity. However, it is possible to vary the number of these constraints which are USEABLE. Variation in the number of USEABLE constraints leads to a change both in the nature of the domain and in the induction runtime and memory requirements. Relaxing a constraint always has the effect of increasing the size of the version space, the run time, and the memory requirements. Like relationship connectivity, the effects of constraint relaxation must be illustrated by means of examples. I investigated constraint relaxation using two variants of the Blocks World from the AI literature that differed only in that a constraint that is USEABLE in one variant is UNUSEABLE in the other. More details are given in Section 4.7.

- *Meta-heuristic.* Since the meta-heuristic is applied after the version-space induction process is completed, varying the meta-heuristic has no effects on the domain states identified. It has only marginal effects on the runtime and memory requirements. By contrast, varying the meta-heuristic can change the domain's state-transition network substantially. Where this results in a change in the number of transition-classes, then there will be a change in the resulting set of planning operators. The most interesting results were observed for the Tank-Farm domain. More details are given in Section 4.8.

In principle, all runs should have been done by inducing a global state-transition network. In practice, it was necessary to perform the HPCE runs by inducing class-states and then merging them and extracting transitions to obtain the global state-transition network. Section 4.9 compares induction of global and class state-transition-networks.

4.2.6 Application to Illustrative Domains: External Standards

Figure 66 tabulates the experiments involving comparison with an external standard. As in Figure 65, there is one row for each domain, and one column for each form of comparison.

Domains	Baseline	External standard		
		Oracle		Expert (Face Validity)
		Output Comparison	Input Comparison	
Piano-playing	P1	-	-	-
Blocks world	B1	B1 and B2	B1, B4, B5	-
HPCE	H1	-	-	H1

Figure 66: Comparing Domains against External Standards.

A full set of experiments has been done only for the Blocks World. No experiments could be done for the Piano-Playing domain, because there are no external standards for the domain. In the absence of an oracle for the HPCE domain, resort has had to be made to face validation using a domain expert. The following experiments were performed:

- *Output Comparison Testing*. Output comparison testing was done for the Blocks World using experiments B1 and B2. More details are in Sections 4.3 and 4.6.

- *Input Comparison Testing*. Input comparison testing was done for the Blocks World using experiments B2, B4, and B5. More details are in Sections 4.7 and 4.10.

- *Face Validation*. The absence of a domain expert for the Piano-Playing domain ruled it out for face validation. Oracles were available for the Blocks World, making face validation unnecessary. There were several domain experts available for the HPCE domain. Face validation was done by one of these experts for baseline experiment H1. More details are in Section 4.3.

4.2.7 Summary of Experiments by Section

Figure 67 summarises the experimental results highlighted in the following sections in this chapter.

Section	Subject	Experiments
4.3	Demonstration	P1, B1, H1
4.4	Within-domain complexity	P3
4.5	Across-domain complexity	*1
4.6	Granularity	B2, P4
4.7	Constraints	B5
4.8	Meta-heuristic	P6
4.9	Partitioning version-space	P3, B3
4.10	Inheritance	B2, B4, H1
4.11	Meta-domain	POI1

Figure 67: Experiments by Section.

4.3 DEMONSTRATION OF INDUCTION

This section demonstrates the induction of planning operators for each of the three illustrative domains.

4.3.1 Detailed Behaviour of POI Algorithm

The Piano-Playing domain is sufficiently simple to be able to demonstrate the behaviour of the full POI algorithm (i.e., both Parts 1 and 2) step-by-step. For conciseness, the domain will be simplified to five `Finger`-instances and one `Key`-instance. The world-state being observed will be assumed to be as described in Figure 68. The required behaviour will be described in the nine steps of the POI algorithm.

```
The index finger of the left hand is pressing the middle C key on
the piano, and the thumb, middle finger, ring finger, and little
finger of the left hand are not pressing any keys.
```

Figure 68: Piano-Playing Example - Observed World-State.

Step (1.1): The first step in Part 1 is to acquire[47] a description of each observed world-state and express it in terms of the relationships that are true in that state. For illustrative purposes, I assume that the POI algorithm acquires the state description given in Figure 69, where:

`finger1` represents the index finger of the left hand,
`finger2` represents the thumb on the left hand,

[47] The precise means of acquisition is irrelevant to this thesis.

finger3 represents the middle finger of the left hand,
finger4 represents the ring finger of the left hand,
finger5 represents the little finger of the left hand,
key1 represents the middle C key on the piano,
pressing represents the fact that a finger is pressing a piano-key, and
notPressing represents the fact that a finger is not pressing any piano-key.

```
[       [pressing finger1 key1]
        [notPressing finger2]
        [notPressing finger3]
        [notPressing finger4]
        [notPressing finger5]     ]
```

Figure 69: Piano-Playing Example - Acquired World-State Description.

Step (1.2): The second step in Part 1 is to recognise the names of objects and relationships from the acquired descriptions. Based on the conventions for the state descriptions, the POI algorithm identifies the following words as naming object-instances in the domain:

```
[finger1, key1, finger2, finger3, finger4, finger5].
```

From its knowledge of these object-instances and its meta-knowledge of the naming conventions (described in Chapter 1), the POI algorithm identifies the object-classes [Finger, Key], with instances [finger1, finger2, finger3, finger4, finger5] and [key1], respectively.

The algorithm recognises one primary relationship-class, named pressing, which is performed by an instance of the Finger object-class on an instance of the Key object-class, i.e., [pressing Finger Key]. The algorithm also recognises the relationship-class [notPressing Finger] as being the absence of the pressing relationship-class, i.e., its *inverse*. In addition, the algorithm recognises that there should be a *converse* relationship-class, i.e., one which expresses the identical relationship but from the Key's viewpoint. The converse relationship would be named [pressedBy Key Finger]. Moreover, the algorithm recognises that the converse relationship-class should itself have an inverse, to be named notPressed.

Step (1.3): The third step in Part 1 is to compile the constraints from the extracted relationship-classes. This is done by pairing all relationship-classes which have at least one object-class in common. In the piano-playing domain, there is only one primary relationship-class: pressing. Pairing pressing with itself guarantees two object-classes in common: Finger and Key. Therefore, two potential binary interrelationship constraints are formed. The first constraint has a common variable formed from the Finger object-class:

Constraint (1):
```
[[pressing ?Finger1 ?Key1] AND
 [pressing ?Finger1 ?Key2]]
```

which would match situations where the same finger is pressing two different piano-keys. The second constraint has a common variable formed from the Key object-class:

Constraint (2):
```
[[pressing ?Finger1 ?Key1] AND
 [pressing ?Finger2 ?Key1]].
```

which would match situations where the same piano-key is being pressed by two different fingers. Constraint (1) would be owned by the `Finger` object-class, and constraint (2) by the `Key` object-class.

In addition, the POI algorithm recognises the need for constraints to express the mutual exclusion between the `pressing` and `pressedBy` relationship-classes and their inverses. These constraints are expressed as the exclusion-rules listed in Figure 70. As none of the four constraints are contradicted by the observed world-state description, they are all marked USEABLE.

```
Exclusion-Rule (1):
IF [[pressing ?Finger1 ?Key1] AND [pressing ?Finger1 ?Key2]]
THEN INVALID.                                                    (USEABLE)

Exclusion-Rule (2):
IF [[pressing ?Finger1 ?Key1] AND [pressing ?Finger2 ?Key1]]
THEN INVALID.                                                    (USEABLE)

Exclusion-Rule (3):
IF [[pressing ?Finger1 ?Key1] AND [notPressing ?Finger1]]
THEN INVALID.                                                    (USEABLE)

Exclusion-Rule (4):
IF [[pressedBy ?Key1 ?Finger1] AND [notPressed ?Key1]]
THEN INVALID.                                                    (USEABLE)
```

Figure 70: Piano-Playing Example - Compiled Exclusion-Rules.

At this point, the POI algorithm has completed the conversion of the observed state description into a domain model consisting of lists of objects, relationships, and interrelationship constraints. This completes the Acquisition Part of the algorithm. It now proceeds with the Induction Part.

Statement	Description	Converse
S_1	[pressing finger1 key1]	[pressedBy key1 finger1]
S_2	[pressing finger2 key1]	[pressedBy key1 finger2]
S_3	[pressing finger3 key1]	[pressedBy key1 finger3]
S_4	[pressing finger4 key1]	[pressedBy key1 finger4]
S_5	[pressing finger5 key1]	[pressedBy key1 finger5]
S_6	[notPressing finger1]	
S_7	[notPressing finger2]	
S_8	[notPressing finger3]	
S_9	[notPressing finger4]	
S_{10}	[notPressing finger5]	
S_{11}	[notPressed key1]	

Figure 71: Piano-Playing Example - State-Description Language.

Step (2.1): As the first step in Part 2, the POI algorithm generates all the valid elementary statements in the state-description language. It does so by instantiating each relationship-class with all the known object-instances that match the relationship's actor and actee object-classes. The instantiated relationship-instances are then tested for non-contradiction of the constraints. In the Piano-Playing domain, the description language generated in this way is as listed in Figure 71. Note that the effect of Steps (1.3) and (2.1) has been to generalise the observed relationship-instance `[pressing finger1 key1]` to the similar relationship-instances involving the other fingers, and to its converses and inverses. Since the converses are simply another way of expressing the primary statements (S_1 to S_5), the converses are not regarded as separate statements in the description language.

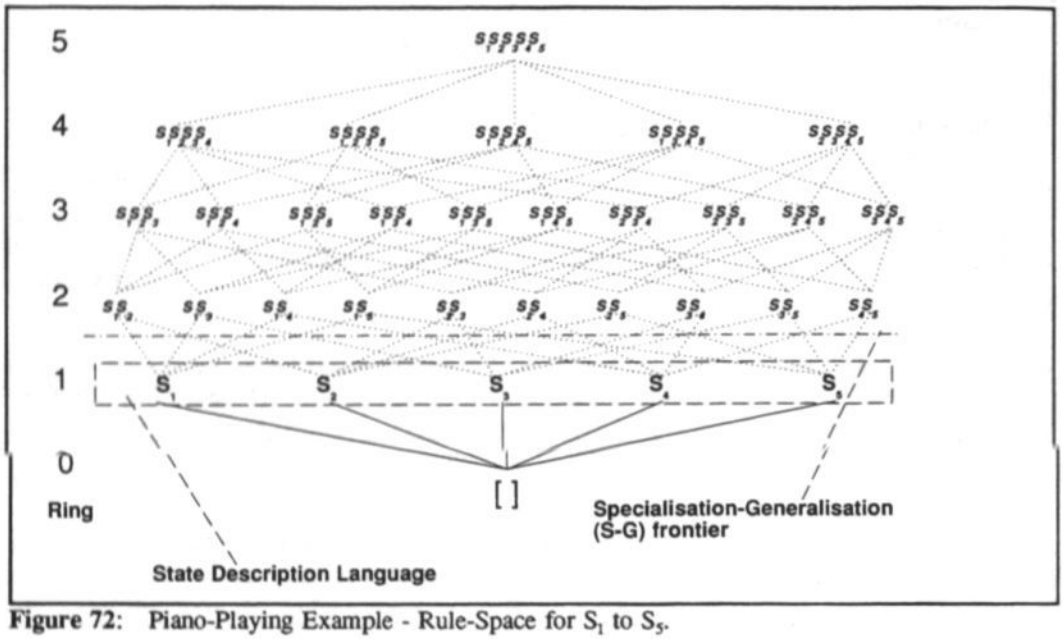

Figure 72: Piano-Playing Example - Rule-Space for S_1 to S_5.

Step (2.2): The second step in Part 2 is the construction of the version space. The description language is used to build the version space, with candidates being eliminated using the exclusion constraints. Figure 72 shows the complete rule-space lattice. For clarity, a lattice containing only the primary relationship-instances, S_1 to S_5, has been drawn[48]. Each node subsumes all the nodes below it in the sub-lattice of which it is the top element. The maximum possible number of lattice nodes is 2^c, where c is the cardinality of the description language. Domain constraints reduce the actual number of lattice nodes built, i.e., they eliminate candidates. Eliminated candidates are shown in italics. The version space is that part of the rule-space lattice lying between the Specialisation-Generalisation (S-G) frontier and the bottom element, i.e., the space of the non-eliminated candidate nodes. The minimum size of the version space is (c + 1). In the Piano-Playing domain, as in most domains, the version-space size lies between (c + 1) and 2^c. When constraints (1) and (2) are both USEABLE and only the primary relationship-class is shown, then the set of nodes immediately below the S-G frontier is the same as the state-description language. The state-description language always appears in ring 1.

Step (2.3): All valid world-state descriptions are now identified in the third step of Part 2 from the version-space nodes that lie just below the S-G frontier. The identified state descriptions are listed in

[48] The full rule-space would have 11 rings and 2048 nodes.

Figure 73.

```
State 1:    [[pressing finger1 key1]
             [notPressing finger2]
             [notPressing finger3]
             [notPressing finger4]
             [notPressing finger5]]

            .
            .

State 5:    [[notPressing finger1]
             [notPressing finger2]
             [notPressing finger3]
             [notPressing finger4]
             [pressing finger5 key1]]

State 6:    [[notPressed key1]
             [notPressing finger1]
             [notPressing finger2]
             [notPressing finger3]
             [notPressing finger4]
             [notPressing finger5]]
```

Figure 73: Piano-Playing Example - Valid State-Descriptions.

Pair of States	Actors	State-Change
State 1 <--> State 2	finger1, finger2	-
State 1 <--> State 3	finger1, finger3	-
State 1 <--> State 4	finger1, finger4	-
State 1 <--> State 5	finger1, finger5	-
State 1 <--> State 6	finger1	pressing <-> notPressing (Transition 1)
State 2 <--> State 3	finger2, finger3	-
State 2 <--> State 4	finger2, finger4	-
State 2 <--> State 5	finger2, finger5	-
State 2 <--> State 6	finger2	pressing <-> notPressing (Transition 2)
State 3 <--> State 4	finger3, finger4	-
State 3 <--> State 5	finger3, finger5	-
State 3 <--> State 6	finger3	pressing <-> notPressing (Transition 3)
State 4 <--> State 5	finger4, finger5	-
State 4 <--> State 6	finger4	pressing <-> notPressing (Transition 4)
State 5 <--> State 6	finger5	pressing <-> notPressing (Transition 5)

Figure 74: Piano-Playing Example - Determining Transitions.

Step (2.4): The fourth step in Part 2 is to determine the set of valid transitions between the world-state

descriptions. By default, the POI algorithm does this using the SA/SSC meta-heuristic. In the Piano-Playing example, the pairs of world-state descriptions are as listed in the left-hand column in Figure 74. The middle and right-hand columns show the results of applying the SA/SSC meta-heuristic. The first ("SA") part of the meta-heuristic tests for exactly one actor. The second ("SSC") part tests that this actor undergoes exactly one state change. Only those transitions passing through the state in which `key1` is not pressed - State 6 - would be considered valid. Figure 75 diagrams the state- and transition-instances as a state-transition network. Note that the transitions are bidirectional.

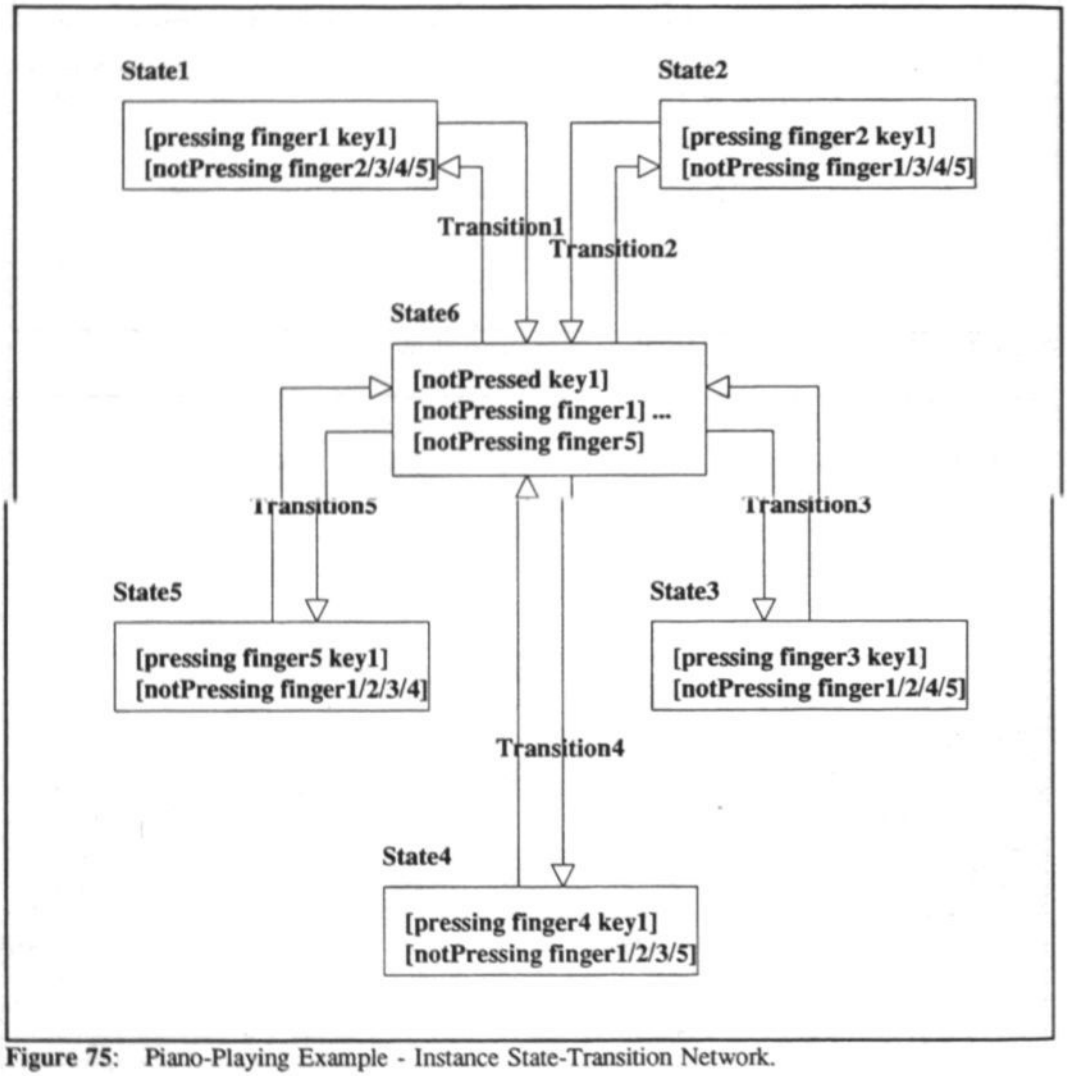

Figure 75: Piano-Playing Example - Instance State-Transition Network.

Step (2.5): The fifth step in Part 2 is to generalise the state descriptions and transitions to obtain state- and transition-classes, respectively. In the Piano-Playing domain, the resulting state- and transition-classes are as shown in Figure 76. Figure 77 diagrams the state- and transition-classes as a state-transition network. Note that the transition-classes are also bidirectional.

```
StateClass1:
              [pressing ?Finger1 ?Key1]
              [notPressing ?Finger2]
              [notPressing ?Finger3]
              [notPressing ?Finger4]
              [notPressing ?Finger5]

StateClass2:
              [notPressed ?Key1]
              [notPressing ?Finger1]
              [notPressing ?Finger2]
              [notPressing ?Finger3]
              [notPressing ?Finger4]
              [notPressing ?Finger5]


TransitionClass1:
              From: StateClass1
              To: StateClass2
```

Figure 76: Piano-Playing Example - State- and Transition-Classes.

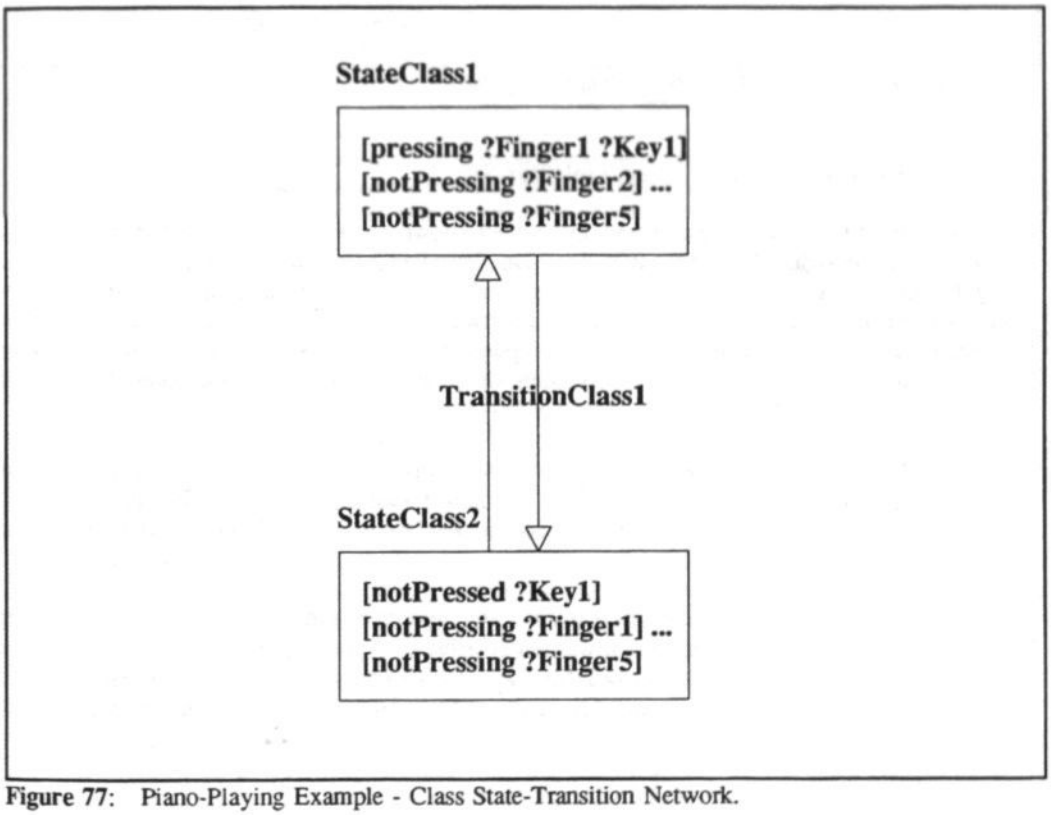

Figure 77: Piano-Playing Example - Class State-Transition Network.

Step (2.6): The final step in Part 2 is to format the transition-classes as STRIPS operators, with preconditions-, add- and delete-lists. The STRIPS operators obtained for the Piano-Playing domain, exactly as output by the single-agent POI implementation, are listed in Figure 78. Note that, in the absence of domain knowledge, the operators have been given arbitrary names by the DUC-ASS program. They could be renamed `lowerOnto` and `raiseFrom`, respectively.

```
POI Step (2.6): EntityManager1 formatting operators ...

Planning operators created:

EntityManager1-operator1 is:
   - my (filter) preconditions are:
      [isFinger ?Finger1]
      [isPianoKey ?PianoKey1]
   - my delete-list is:
      [notPressed ?PianoKey1]
      [notPressing ?Finger1]
   - my add-list is:
      [pressing ?Finger1 ?PianoKey1]

EntityManager1-operator2 is:
   - my (filter) preconditions are:
      [isFinger ?Finger1]
      [isPianoKey ?PianoKey1]
   - my delete-list is:
      [pressing ?Finger1 ?PianoKey1]
   - my add-list is:
      [notPressed ?PianoKey1]
      [notPressing ?Finger1]

Planning operators formatted; POI Step (2.6) completed.
```

Figure 78: Piano-Playing Example - Planning Operators.

4.3.2 Reproducing Nilsson's (1980) Operator-Set

The Blocks World is used to demonstrate that a well-known set of planning operators from the AI literature can be reproduced using the POI algorithm. Only Part 2 (Induction) of the algorithm is used, taking the entity-relationship model of the Blocks World as its input. The minimum number of object-instances needed to reproduce Nilsson's (1980) operator-set is one hand, two blocks, and one table. The domain model tabulated in Figure 79 was input to the DUC-ASS program. Given this domain model, one might expect that the POI algorithm would induce the five world-states described in Figure 80.

In fact, the DUC-ASS program induced 12 world-states. Inspection shows that, in addition to the five expected world-states, other states were induced in which at least one block was "floating in mid-air". An example "floating-block" world-state, as induced by DUC-ASS, is described and depicted in Figure 81.

The reason for this unexpected behaviour is that the POI algorithm (and its implementations) has been limited to binary constraints. Therefore, it was not possible to express in the input domain model the workings of gravity. The effect of gravity would be such as to require a block to be either held by a hand, or supported by a table, or supported by another block. This would require the triple constraint:

```
IF [notHeld ?Block1]
AND [notOn ?Block1]
AND [notOnTable ?Block1]
THEN INVALID.
```

Object-Classes:

- `Hand`, with instance `hand1`.
- `Block`, with instances `block1` and `block2`.
- `Table`, with instance `table1`.

Relationship-Classes:

- `holding` which relates `Hand` instances to `Block` instances. Its converse is `heldBy`. Their inverses are `notHolding` and `notHeld`, respectively.
- `on` which relates `Block` instances to `Block` instances. Its converse is `beneath`. Their inverses are `notOn` and `notBeneath`, respectively.
- `onTable` which relates `Block` instances to `Table` instances. Its converse is `supporting`. Their inverses are `notOnTable` and `notSupporting`, respectively.

Constraints:

- The same hand cannot hold two or more blocks simultaneously.
- The same block cannot be held by two or more hands simultaneously.
- A hand cannot be both holding a block and not holding a block simultaneously.
- A block cannot be both being held by a hand and not being held by a hand simultaneously.
- A block cannot be on top of itself.
- The same block cannot be on top of two other blocks simultaneously.
- The same block cannot be beneath two other blocks simultaneously.
- Two blocks cannot be mutually on top of each other.
- The same block cannot be both on top of another block and held by a hand.
- The same block cannot be both beneath another block and held by a hand.
- A block cannot be both on top of another block and not on top of another block simultaneously.
- A block cannot be both beneath another block and not beneath any other block simultaneously.
- The same block cannot be on two or more tables simultaneously.
- A block cannot be both on a table and held by a hand simultaneously.
- The same block cannot be both on top of another block and on a table.
- A block cannot be both on a table and not on a table simultaneously.
- A table cannot be both supporting a block and not supporting any block simultaneously.

Figure 79: Two-Blocks World - Domain Model.

The POI algorithm is not at fault. On the contrary, it has induced correctly the set of states for a two-Blocks World in which there is no gravity, as would be found in a spacecraft in orbit around the Earth. Without gravity, blocks can float[49].

We are concerned in this section with demonstrating that the POI algorithm can reproduce Nilsson's (1980) operator-set. His state-transition network did not depict states in which blocks were floating. The conclusion must be that Nilsson implicitly assumed an Earth-bound Blocks World. Therefore, the problem lies with the domain model, which is an imperfect representation of what Nilsson intended. Similar imperfections are present in the models of the Dining Philosophers, Aircraft Scheduling, and HPCE domains. Taking the Blocks World as an example domain, Section 4.10 shows how the introduction of inheritance enables the domain model to be refined so as to eliminate such

[49] In practice, the states in which any block is on the table or on another block would rarely be observed in orbit. The contact forces between any objects in contact would be sufficient to cause them to drift apart. In manned space missions, Velcro is often used where it is necessary to keep objects in contact.

imperfections.

State 1:

```
[[notHolding hand1] [onTable block1 table1] [onTable block2 table1]
 [notHeld block1] [notHeld block2] [notOn block1] [notOn block2]
 [notBeneath block1] [notBeneath block2]]
```

State 2:

```
[[holding hand1 block1] [onTable block2 table1]
 [notHeld block2] [notOn block1] [notOn block2]
 [notOnTable block1] [notBeneath block2]]
```

State 3:

```
[[holding hand1 block2] [onTable block1 table1]
 [notHeld block1] [notOn block1] [notOn block2]
 [notBeneath block1] [notOnTable block2]]
```

State 4:

```
[[notHolding hand1] [on block1 block2] [onTable block2 table1]
 [notHeld block1] [notHeld block2] [notOn block2]
 [notBeneath block1] [notOnTable block1]]
```

State 5:

```
[[notHolding hand1] [on block2 block1] [onTable block1 table1]
 [notHeld block1] [notHeld block2] [notOn block1]
 [notBeneath block2] [notOnTable block2]]
```

Figure 80: Two-Blocks World - Five Expected World-States.

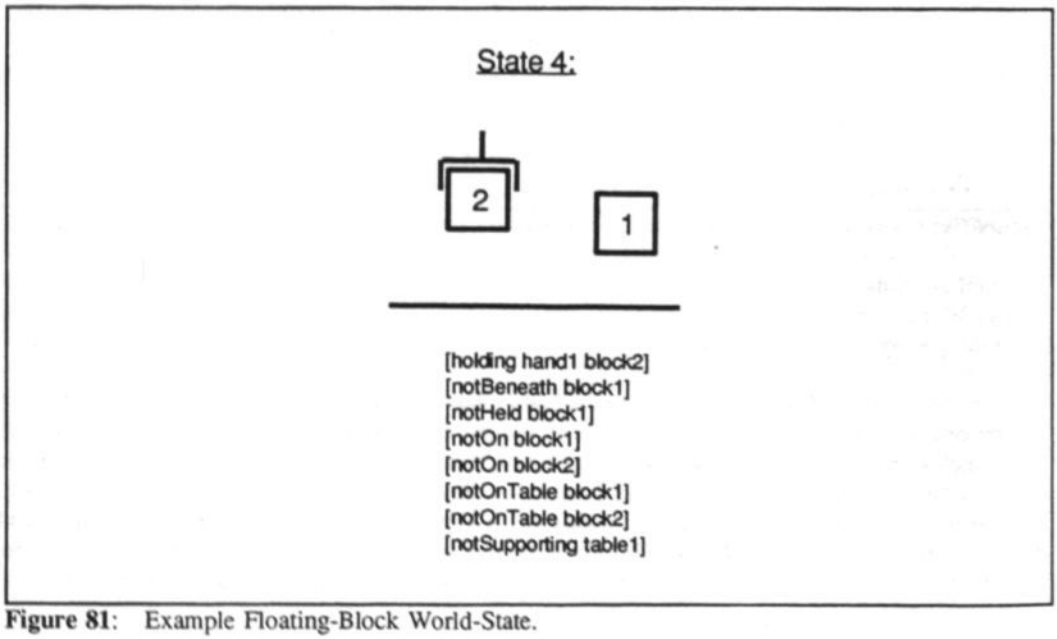

Figure 81: Example Floating-Block World-State.

```
EntityManager1-operator1 is:
   - my (filter) pre-conditions are:
      [isHand ?Hand1]
      [isBlock ?Block1]
      [isTable ?Table1]
      [isBlock ?Block2]
      [notHeld ?Block2]
      [notOn ?Block2]
      [notOnTable ?Block1]
      [onTable ?Block2 ?Table1]
   - my delete-list is:
      [notBeneath ?Block1]
      [notHeld ?Block1]
      [notHolding ?Hand1]
      [on ?Block1 ?Block2]
   - my add-list is:
      [holding ?Hand1 ?Block1]
      [notBeneath ?Block2]
      [notOn ?Block1]

EntityManager1-operator2 is:
   - my (filter) pre-conditions are:
      [isHand ?Hand1]
      [isBlock ?Block1]
      [isTable ?Table1]
      [isBlock ?Block2]
      [notHeld ?Block2]
      [notOn ?Block2]
      [notOnTable ?Block1]
      [onTable ?Block2 ?Table1]
   - my delete-list is:
      [holding ?Hand1 ?Block1]
      [notBeneath ?Block2]
      [notOn ?Block1]
   - my add-list is:
      [notBeneath ?Block1]
      [notHeld ?Block1]
      [notHolding ?Hand1]
      [on ?Block1 ?Block2]
```

Figure 82: Two-Blocks World - **Unstack** and **Stack**.

Based on this conclusion, all the induced states containing floating blocks were marked UNUSEABLE, and the remaining steps of the POI algorithm were completed. Four operators are induced. Comparison with Nilsson's (1980) operator-set shows that:

- Operators 1 to 4 correspond to **`unstack`** and **`stack`** (see Figure 82), and **`pickup`** and **`putdown`** (see Figure 83).

- The delete- and add-lists are identical to those listed in Nilsson (1980, p. 281), when the baseline relationship-variables are replaced by the equivalents in Nilsson's representation (as defined in Figure 57).

- The induced operators include filter preconditions which are not to be found in Nilsson's (1980) operator-set. Inspection shows that:

  - Some of the preconditions test for class-membership, e.g., `[isBlock ?Block2]`. Charniak and McDermott (1985) use the predicate `is` for the same purpose, e.g., `(is ?y block)`, as in (*ibid.*, Figure 9.8, p. 493). Since Nilsson (1980) had variables only for blocks, he did not need class-membership tests.

  - Other filter preconditions are baseline relationship-variables which have no

equivalent in Nilsson's (1980) representation.

- The remaining filter preconditions are an artefact of the two-blocks world, as comparison with one- and three-blocks world runs shows. For example, **operator1** includes [onTable ?Block2 ?Table1] as a filter precondition. The precondition that the block beneath the one being unstacked should be on the table is specific to the two-blocks world. In the three-blocks world, the block being unstacked may be on the top of a stack of three blocks.

```
EntityManager1-operator3 is:
   - my (filter) pre-conditions are:
      [isHand ?Hand1]
      [isBlock ?Block1]
      [isTable ?Table1]
      [isBlock ?Block2]
      [notOn ?Block1]
      [onTable ?Block2 ?Table1]
   - my delete-list is:
      [notBeneath ?Block1]
      [notHeld ?Block1]
      [notHolding ?Hand1]
      [onTable ?Block1 ?Table1]
   - my add-list is:
      [holding ?Hand1 ?Block1]
      [notOnTable ?Block1]

EntityManager1-operator4 is:
   - my (filter) pre-conditions are:
      [isHand ?Hand1]
      [isBlock ?Block1]
      [isTable ?Table1]
      [isBlock ?Block2]
      [notOn ?Block1]
      [onTable ?Block2 ?Table1]
   - my delete-list is:
      [holding ?Hand1 ?Block1]
      [notOnTable ?Block1]
   - my add-list is:
      [notBeneath ?Block1]
      [notHeld ?Block1]
      [notHolding ?Hand1]
      [onTable ?Block1 ?Table1]
```

Figure 83: Two-Blocks World - **Pickup** and **Putdown**.

4.3.3 Inducing Operators for Novel Domain

The HPCE domain is used to demonstrate the ability of the POI algorithm to induce operators for a complex, real-world domain not found in the AI literature. Only Part 2 (Induction) of the algorithm is used, taking the entity-relationship model of the HPCE domain as its input.

The complexity of the HPCE domain was such that it was not possible to induce a global state-transition network within the available computer memory. It was necessary to employ two countermeasures in combination: version-space partitioning, and inheritance. These countermeasures are described in Sections 4.9 and 4.10, respectively. Moreover, it was necessary to restrict the runs to one EntityInstance per EntityClass.

```
EntityManager1-operator103 is:
   - my (filter) pre-conditions are:
      [isCartridge ?Cartridge1]
      [isCapillary ?Capillary1]
      [notPowered ?Capillary1]
      [notShinedOn ?Capillary1]
      [notTransmittingTo ?Capillary1]
      [notContaining ?Capillary1]
      [notFittingTo ?Cartridge1]
   - my delete-list is:
      [notHeld ?Capillary1]
      [notHolding ?Cartridge1]
   - my add-list is:
      [holding ?Cartridge1 ?Capillary1]

EntityManager1-operator17 is:
   - my (filter) pre-conditions are:
      [isReceptacle ?Receptacle1]
      [isSide ?Side1]
      [isVial ?Vial1]
      [isInstrumentBody ?InstrumentBody1]
      [receiving ?Receptacle1 ?Vial1]
      [fittingTo ?Receptacle1 ?InstrumentBody1]
      [notOnOutlet ?Receptacle1]
   - my delete-list is:
      [notOnInlet ?Receptacle1]
   - my add-list is:
      [onInlet ?Receptacle1 ?Side1]
      [notOnOutletBy ?Side1]
```

Figure 84: HPCE Instrument-Assembly Operators.

Using both countermeasures, a set of 362 planning operators was obtained in 2127 seconds of runtime. Because no check has been implemented in DUC-ASS whether a newly-created planning operator duplicates one that has been previously created early in the run, there were many replicated operators. Inspection showed that the induced operators could be grouped into operators modelling:

- *assembling the instrument*. For example, **`operator103`** models placing a capillary into a cartridge, and **`operator17`** models the fitting of a receptacle to an instrument body on the inlet side. See Figure 84.

- *preparing the instrument for use*. For example, **`operator71`** models placing a vial into a tray. See Figure 85.

```
EntityManager1-operator71 is:
   - my (filter) pre-conditions are:
      [isTray ?Tray1]
      [isVial ?Vial1]
      [notPressurised ?Vial1]
      [notReceived ?Vial1]
      [notContaining ?Vial1]
      [notFittingTo ?Tray1]
   - my delete-list is:
      [notRetained ?Vial1]
      [notRetaining ?Tray1]
   - my add-list is:
      [retaining ?Tray1 ?Vial1]
```

Figure 85: HPCE Instrument-Preparation Operator.

```
EntityManager1-operator11 is:
   - my (filter) pre-conditions are:
      [isReceptacle ?Receptacle1]
      [isVial ?Vial1]
      [isTray ?Tray1]
      [isInstrumentBody ?InstrumentBody1]
      [isSide ?Side1]
      [notPressurised ?Vial1]
      [retaining ?Tray1 ?Vial1]
      [notContaining ?Vial1]
      [fittingTo ?Receptacle1 ?InstrumentBody1]
      [notOnInlet ?Receptacle1]
      [onOutlet ?Receptacle1 ?Side1]
   - my delete-list is:
      [notReceived ?Vial1]
      [notReceiving ?Receptacle1]
   - my add-list is:
      [receiving ?Receptacle1 ?Vial1]

EntityManager1-operator9 is:
   - my (filter) pre-conditions are:
      [isDetector ?Detector1]
      [isOn ?On1]
      [isInstrumentBody ?InstrumentBody1]
      [isPowerSupply ?PowerSupply1]
      [fittingTo ?Detector1 ?InstrumentBody1]
      [notDetecting ?Detector1]
      [notGenerating ?Detector1]
      [inState ?PowerSupply1 ?On1]
   - my delete-list is:
      [notInState ?Detector1]
   - my add-list is:
      [inState ?Detector1 ?On1]
```

Figure 86: HPCE Analysis Performance Operators.

- *performing an analysis*. For example, **operator11** models the raising of a vial to a receptacle, and **operator9** models switching on a detector. See Figure 86.

Since the HPCE domain was restricted to one EntityInstance per EntityClass, it was not possible to induce a state-transition network representing the complete canonical analysis run (i.e., PRE-RINSE, INJECT, SEPARATE, and POST-RINSE). The minimum number of EntityInstances needed to do so would have had to include:

- two Receptacles, one on the inlet side and the other on the outlet side, and

- six Vials, comprising one empty ("waste") vial, one vial containing regenerator solution, one containing de-ionised water, one containing the sample to be analysed, and two vials containing buffer solution (one at the inlet receptacle and the other at the outlet).

Despite this restriction, the baseline HPCE run resulted in the induction of a set of planning operators. As no planning operators are given in the Beckman user documentation, all are novel. Moreover, the induced operator-set demonstrated conclusively the advantage of the POI algorithm over other operator-learning techniques in that the POI algorithm:

- *induced operators that could not have been acquired from the canonical plan*. For example the instrument-assembly and -preparation operators cannot be obtained from the canonical plan.

- *decomposed the operators in the canonical plan partly into more primitive operators*. For

example, raising of a vial to a receptacle is a primitive action of both **rinse** and **inject**, and switching on the detector is a primitive action of **separate**.

Finally, a domain expert has assessed the induced operators as being representative of the behaviour of the Beckman HPCE analyser (Eckhard, 1995).

4.4 WITHIN-DOMAIN COMPLEXITY

In the previous chapter, a theoretical analysis was done of the complexity of the POI algorithm. The contribution of each RelationClass to the domain description language was expressed in terms of the numbers of ObjectInstances taking part in the relationship. A number of conclusions can be drawn for a comparison of the theoretical analysis with an empirical complexity analysis. The Piano-Playing domain is used for comparison.

In the Piano-Playing domain, there is just one RelationClass, which joins the `Finger` and `Key` ObjectClasses. The number of statements in the Piano-Playing domain description language is:

$$i_{Finger} * i_{Key} + i_{Finger} + i_{Key},$$

where i_{Finger} and i_{Key} are the numbers of instances of the `Finger` and `Key` object-classes, respectively. The theoretical maximum complexity of the version-space construction will then be 2 to the power of the number of statements. Since the domain constraints are used to prune the version-space ("candidate elimination"), this maximum complexity will not be reached. Figure 87 shows the sizes of the version space (in numbers of lattice nodes) induced for a range of numbers of `Fingers` and `Keys`.

Fingers/Keys	1	2	3	4	5
1	5	12	28	64	144
2	12	34	92	240	608
3	28	92	286	848	2416
4	64	240	848	2840	9072
5	144	608	2416	9072	

Figure 87: Actual Complexity of Piano-Playing Version-Space (in nodes).

It is important to know if the actual complexity remains non-polynomial. Regression analysis of the size of the version space against the number of `Fingers` is consistent with exponential behaviour, giving a Pearson correlation coefficient of 0.9997 (or greater) and a standard error of estimate of $4 * 10^{-2}$ (or less) for 1, 2 and 3 `Keys`. Similar results are obtained for other parameters (i.e., runtime (see Figure 88) and memory used (see Figure 89)), as well as for the Blocks World. Therefore, I conclude that the POI algorithm suffers from a combinatorial explosion for within-domain complexity, confirming the theoretical complexity analysis in Section 3.6.1.

Fingers/Keys	1	2	3	4	5
1	0.9	1.5	2.3	4.4	15.8
2	1.5	3.6	8.8	27.0	129.5
3	2.3	8.8	40.6	229.9	1622.4
4	4.4	27.0	229.9	2300.3	
5	15.8	129.5	1622.4		

Figure 88: Runtime for Piano-Playing Domain (in seconds).

The net effect of the combinatorially-explosive nature of the POI algorithm is that there comes a point at which a realistically-large domain cannot be modelled on present-day PCs. Figure 89 demonstrates clearly that it would not be possible to induce operators for the full-size Piano-Playing domain, with its 10 fingers and 88 piano-keys. The bottleneck is memory usage, rather than runtime. Sections 4.9 and 4.10 describe countermeasures that delay the point at which available memory becomes a limitation on the complexity of domain that can be modelled.

Fingers/Keys	1	2	3	4	5
1	37.9	49.7	64.5	91.6	146.9
2	49.7	74.2	130.0	244.2	502.3
3	64.5	130.0	309.8	767.0	1928.7
4	91.6	244.2	767.0	2422.8	
5	146.9	502.3	1928.7		

Figure 89: Memory Usage for Piano-Playing Domain (in kilobytes).

4.5 ACROSS-DOMAIN COMPLEXITY

Across-domain complexity has been assessed by analysing a series of runs, one for each of the following domains:

- Finger-Crossing (FC),
- Piano-Playing (PP),
- Tank-Farm (TF),
- Blocks World (Genesereth and Nilsson (1987) variant) (BWGN),

- Blocks World (Nilsson (1980) variant) (BWN),
- Dining Philosophers (DP),
- Aircraft Scheduling (AS),
- POI, and
- HPCE.

To minimise the effects of within-domain complexity, each domain was run for the case where each object-class in the domain had just one instance. This led to distortion in the Finger-Crossing domain where no operators could be induced. The results are tabulated in Figure 90. The following abbreviations are used for the measures:

- "OC" means number of object-classes,
- "RC" means number of relationship-classes,
- "RI" means number of relationship-instances,
- "FOM" means figure of merit, i.e., the theoretical maximum complexity divided by the actual complexity, where complexity is measured in numbers of nodes, and
- "Y" and "N" stand for "Yes" and "No".

Measure	FC	PP	TF	BWGN	BWN	DP	AS	POI	HPCE
OC	1	2	2	2	3	5	9	19	24
RC	1	1	2	2	3	5	10	25	18
RI	3	3	6	6	9	13	10	73	42
Max nodes	8	8	64	64	512	32768	2^{30}	2^{73}	2^{42}
Actual nodes	4	5	24	20	88	1080	201	*	724
FOM	2.0	1.6	2.7	3.2	5.8	30.3	5E6	*	6E9
States	1	2	3	2	3	3	72	*	129
Transitions	0	1	2	1	2	2	172	*	181
Operators	0	2	4	2	2	4	344	*	362
Inheritance	N	N	N	N	N	N	N	Y	Y
Class-state merging	N	N	N	N	N	N	Y	Y	Y
Runtime (secs)	0.3	0.9	1.9	1.6	5.6	257.3	229.9	*	2127.4
Memory (KB)	77.3	38.4	72.7	75.3	142.8	833.5	1201.2	*	3916.8

* Could not be assessed; see Section 4.11.

Figure 90: Across-Domain Complexity Results.

Regression analysis was done of the numbers of object-classes, relationship-classes, and relationship-instances against the size of the version space (in actual nodes), the runtime (in seconds), and the memory usage (in kilobytes). Only those domains for which operators were obtained without using any countermeasures were included in the analysis.

Third-order regression gave the highest coefficient of correlation in all cases, with standard errors of estimation that ranged from acceptable to excellent. Therefore, I conclude that across-domain complexity can be regarded as consistent with a third-order model, i.e., polynomial behaviour.

It is noteworthy that across-domain complexity is less severe than within-domain complexity.

4.6 VARYING DOMAIN GRANULARITY

There are three aspects to domain granularity:

- *the number of object-classes.* The Blocks World is used to investigate the effects of varying the number of object-classes.

- *the number of relationship-classes.* The Piano-Playing domain is used to investigate the effects of varying the number of relationship-classes.

- *the connectivity between object- and relationship-classes.* Connectivity is investigated by comparing the Tank-Farm domain with the Genesereth and Nilsson (1987) variant of the Blocks World.

```
EntityManager1-operator3 is:
   - my (filter) pre-conditions are:
      [isBlock ?Block3]
      [isBlock ?Block1]
      [isBlock ?Block2]
      [isTable ?Table1]
      [notBeneath ?Block3]
      [notOn ?Block1]
      [notOn ?Block2]
      [notOnTable ?Block3]
      [onTable ?Block1 ?Table1]
      [onTable ?Block2 ?Table1]
   - my delete-list is:
      [notBeneath ?Block1]
      [on ?Block3 ?Block2]
   - my add-list is:
      [notBeneath ?Block2]
      [on ?Block3 ?Block1]

EntityManager1-operator5 is:
   - my (filter) pre-conditions are:
      [isBlock ?Block2]
      [isBlock ?Block4]
      [isTable ?Table1]
      [isBlock ?Block1]
      [notBeneath ?Block2]
      [notOn ?Block4]
      [onTable ?Block1 ?Table1]
      [onTable ?Block4 ?Table1]
   - my delete-list is:
      [notBeneath ?Block4]
      [notOn ?Block2]
      [onTable ?Block2 ?Table1]
   - my add-list is:
      [notOnTable ?Block2]
      [on ?Block2 ?Block4]
```

Figure 91: **S** and **M** Operators for 3-Blocks Genesereth and Nilsson (1987) World.

4.6.1 Varying Object-Classes

Section 4.1.4 identified variants of the Blocks World, other than Nilsson (1980), that differed in the numbers of object-classes. These Blocks World variants could - in principle - be used to analyse POI's sensitivity to varying object-classes by reproducing the different operator-sets. In practice, the Ramsay and Barratt (1987) variant is unsuitable, because it introduces the metric relationship of `position`. There are no difficulties in reproducing the Genesereth and Nilsson (1987) variant.

The Genesereth and Nilsson (1987) representation was reproduced using the DUC-ASS program. A three-blocks world is needed to induce the S and M operators. Like the Nilsson (1980) representation, states were induced in which blocks "floated" in mid-air. These floating-block states were marked UNUSEABLE. The resulting operators equivalent to the S and M operators are shown in Figure 91. As for the experiment to reproduce Nilsson's (1980) operator-set (see Section 4.3.2), comparison shows that:

- The delete- and add-lists are identical, when the baseline relationship-variables are replaced by the equivalents in Genesereth and Nilsson's (1987) representation.

- The induced operators include filter preconditions that are not to be found in Genesereth and Nilsson's (1987) operator-set. The reasons are as listed in Section 4.3.2.

Measure	Piano-Playing	Tank-Farm
1-Finger, 1-Key versus 1-Tank, 1-Reactor		
States	2	3
Transitions	1	2
Operators	2	4
1-Finger, 2-Key versus 1-Tank, 2-Reactor		
States	3	5
Transitions	2	4
Operators	2	4
2-Finger, 2-Key versus 2-Tank, 2-Reactor		
States	7	17
Transitions	8	24
Operators	2	6

Figure 92: Results of Adding Relationship-Class.

4.6.2 Varying Relationship-Classes

In the Piano-Playing domain, the two object-classes are linked by a single relationship-class. This was varied in experiment P4 by adding a second relationship-class between the two object-classes. The result was a domain with different characteristics: the Tank-Farm domain.

Three pairs of runs were done in experiment P4. In the first pair, a one-Finger, one-Key Piano-Playing domain was compared with a one-Tank, one-Reactor Tank-Farm domain. In the second pair, a one-Finger, two-Key Piano-Playing domain was compared with a one-Tank, two-Reactor Tank-Farm domain. In the third pair, a two-Finger, two-Key Piano-Playing domain was compared with a two-Tank, two-Reactor Tank-Farm domain. The results are summarised in Figure 92. Figure 93 shows the difference in behaviour of the two domains by comparing their state-transition networks for the second pair of runs. I conclude that increasing the number of relationship-classes leads to an increase in the numbers of states, transitions, and operators induced.

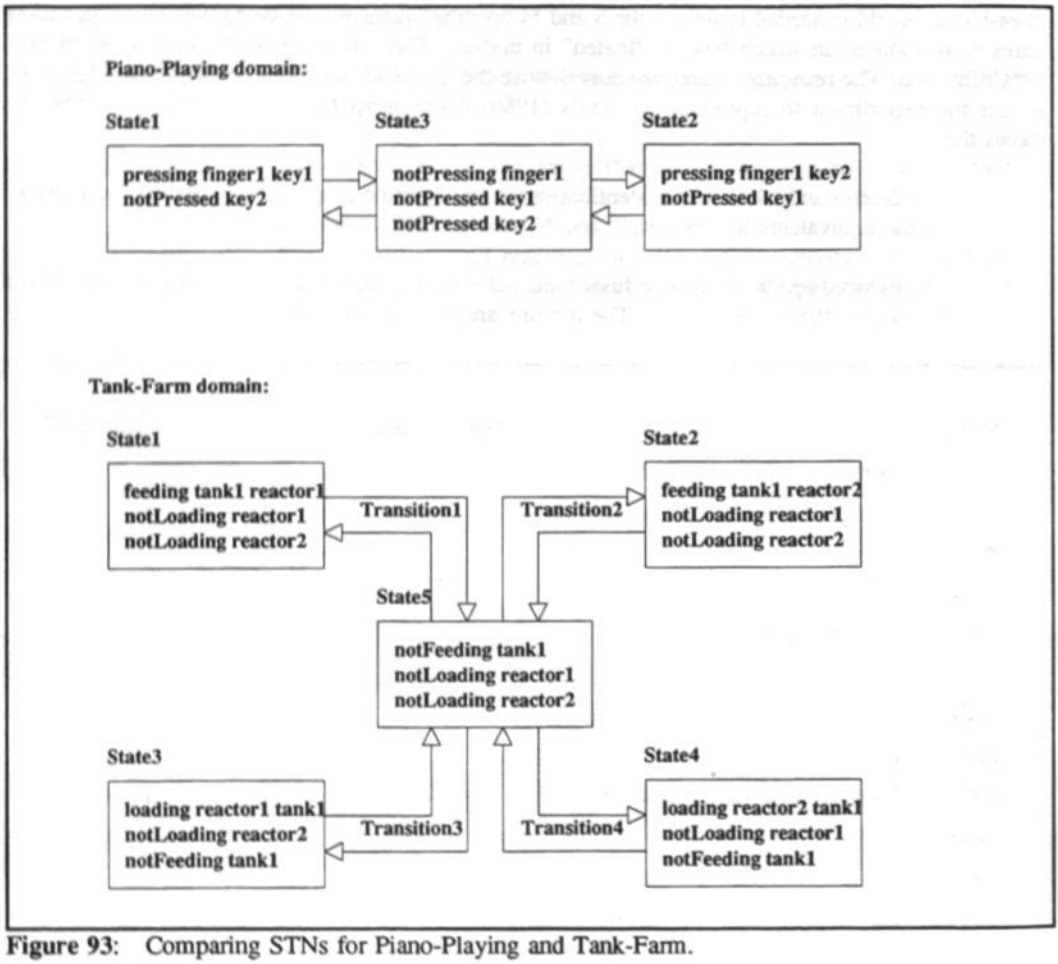

Figure 93: Comparing STNs for Piano-Playing and Tank-Farm.

4.6.3 Varying Connectivity

The Tank-Farm domain shares with the Genesereth and Nilsson (1987) variant of the Blocks World the characteristics of having two object-classes and two relationship-classes. However, the connectivity differs.

The effect of varying connectivity was sampled by comparing three pairs of runs. In the first pair, a one-Tank, one-Reactor Tank-Farm domain was compared with a one-Block, one-Table Blocks

World. In the second pair, a one-`Tank`, two-`Reactor` Tank-Farm domain was compared with a two-`Block`, one-`Table` Blocks World. In the third pair, a two-`Tank`, two-`Reactor` Tank-Farm domain was compared with a two-`Block`, two-`Table` Blocks World.

Measure	Tank-Farm	Blocks World, Genesereth and Nilsson (1987) variant
1-Tank, 1-Reactor versus 1-Block, 1-Table		
States	3	1
Transitions	2	0
Operators	4	0
1-Tank, 2-Reactor versus 2-Block, 1-Table		
States	5	3
Transitions	4	2
Operators	4	2
2-Tank, 2-Reactor versus 2-Block, 2-Table		
States	17	8
Transitions	24	15
Operators	6	8

Figure 94: Results of Varying Connectivity.

The results are summarised in Figure 94. The operators induced for the Tank Farm domain model the starting and stopping of feeding and loading. In the 2-tank, 2-reactor variant, two more operators are induced, modelling the direct transitions from loading to feeding and from feeding to loading. Operators are induced for the Genesereth and Nilsson (1987) Blocks World only when there are at least two blocks. Their **`move`** operator is only induced when there are at least three blocks. The two-block, two-table variant also results in operators, not reported in Genesereth and Nilsson (1987), for stacking, unstacking and moving blocks between tables. Additionally, an operator is induced that represents the moving of complete stacks between tables. This indicates that the domain model is imperfect, in that it lacks a higher-order constraint that takes effect only when there are multiple tables.

Figure 95 shows the difference in behaviour of the two domains by comparing the state-transition networks for the second pair of runs. In the Blocks World, transitions going outwards from State5 are instantiations of Genesereth and Nilsson's (1987) **`stack`** operator, and those going inwards are **`unstack`** instantiations. I conclude that varying the connectivity of relationship-classes has a major effect on the numbers of states, transitions, and operators induced. From Section 3.6.1 it can be seen that the effect depends on the respective numbers of instances of the object-classes linked by the relationship-classes.

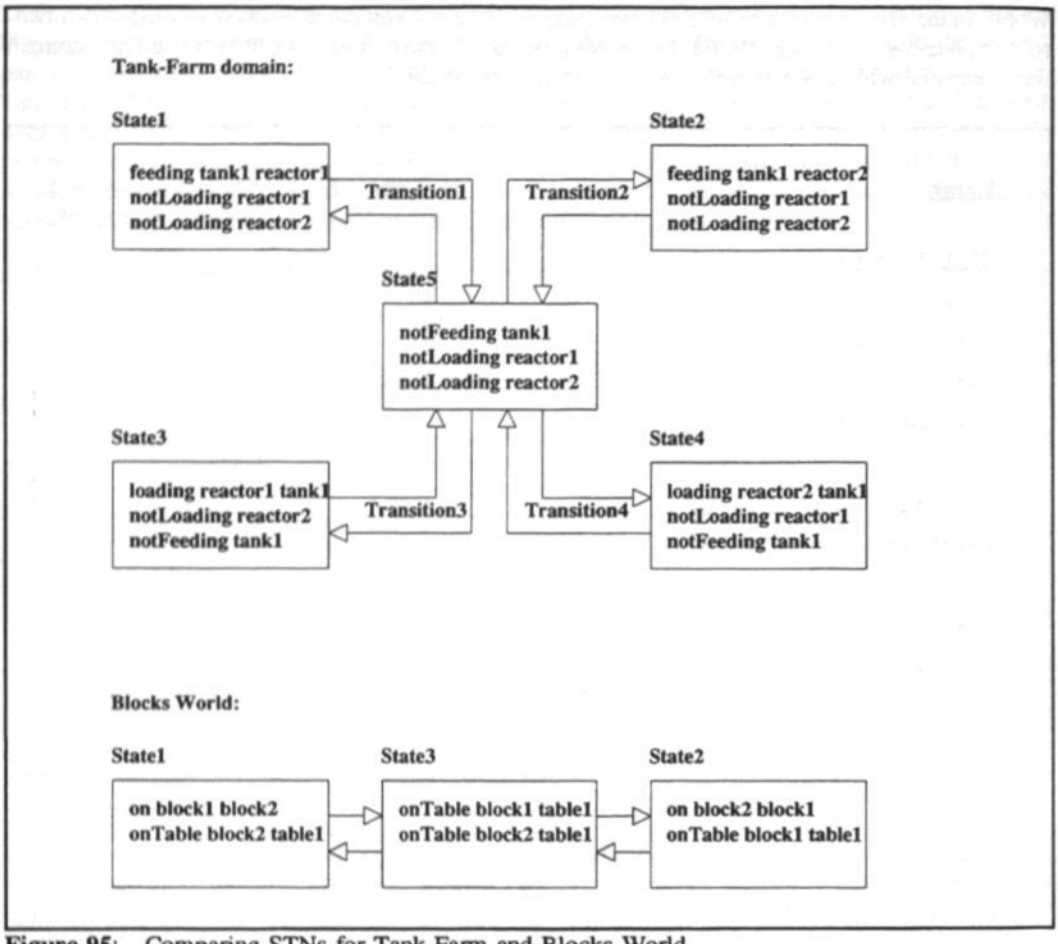

Figure 95: Comparing STNs for Tank-Farm and Blocks World.

4.7 VARYING DOMAIN CONSTRAINTS

An indication of the effects of varying domain constraints can be illustrated using the Nilsson (1980) variant of the Blocks World. In Section 4.5.1, it was noted that the Nilsson variant was a member of the three-object-classes group, which also contained Rich and Knight (1991) and Thornton and du Boulay (1992). Each group member modeled the blocks world using three object-classes and three relationship-classes, and identified four operators.

Although they belong to the same group, there is a subtle difference between the Nilsson's (1980) operator-set and the operator-set of Rich and Knight (1991). In Rich and Knight's operator-set, the predicate `CLEAR(x)` is not deleted either when a block is picked up (see Figure 96) or when a block is unstacked. Similarly, the predicate `CLEAR(x)` is not added when blocks are put down or unstacked. The net effect is that a block that is held by a hand is still `CLEAR`. By contrast, Nilsson deletes the predicate *`CLEAR(x)`* whenever a block is held in the hand. In essence, the exclusion-rule:

```
IF [holding ?Hand1 ?Block1] AND [clear ?Block1] THEN INVALID
```

is USEABLE in Nilsson's blocks world, but UNUSEABLE in that of Rich and Knight.

Nilsson (1980) representation:

```
pickup(x)
   P & D: ONTABLE(x), CLEAR(x), HANDEMPTY
   A: HOLDING(x)
```

Rich and Knight (1991) representation:

```
PICKUP(x)
   P: CLEAR(x), ONTABLE(x), ARMEMPTY
   D: ONTABLE(x), ARMEMPTY
   A: HOLDING(x)
```

Figure 96: Comparing **Pickup** for Nilsson (1980) and Rich and Knight (1991).

```
EntityManager1-operator3 is:
   - my (filter) pre-conditions are:
      [isHand ?Hand1]
      [isBlock ?Block1]
      [isTable ?Table1]
      [isBlock ?Block2]
      [notBeneath ?Block1]
      [notOn ?Block1]
      [onTable ?Block2 ?Table1]
   - my delete-list is:
      [notHeld ?Block1]
      [notHolding ?Hand1]
      [onTable ?Block1 ?Table1]
   - my add-list is:
      [holding ?Hand1 ?Block1]
      [notOnTable ?Block1]

EntityManager1-operator1 is:
   - my (filter) pre-conditions are:
      [isHand ?Hand1]
      [isBlock ?Block1]
      [isBlock ?Block2]
      [isTable ?Table1]
      [notBeneath ?Block1]
      [notHeld ?Block2]
      [notOn ?Block2]
      [notOnTable ?Block1]
      [onTable ?Block2 ?Table1]
   - my delete-list is:
      [notHeld ?Block1]
      [notHolding ?Hand1]
      [on ?Block1 ?Block2]
   - my add-list is:
      [holding ?Hand1 ?Block1]
      [notBeneath ?Block2]
      [notOn ?Block1]
```

Figure 97: **Pickup** and **Stack** with Constraint made UNUSEABLE.

The POI algorithm was used to reproduce the Rich and Knight (1991) operator-set by making the exclusion-rule UNUSEABLE. Figure 97 shows the equivalents of the `pickup` and `stack` operators induced. As for the experiment to reproduce Nilsson's (1980) operator-set (see Section 4.3.2), comparison shows that:

- The delete- and add-lists are identical, when the baseline relationship-variables are replaced by the equivalents in Rich and Knight's (1991) representation.

- The induced operators include filter preconditions that are not to be found in Rich and Knight's (1991) operator-set. The reasons are as listed in Section 4.3.2.

Adopting Rich and Knight's (1991) representation has no consequences for a blocks world in which there is a single hand and where the initial and goal states are correctly described. However, problems would arise if Rich and Knight's operator-set were to be used when:

- *there were multiple hands.* One hand could "snatch" a block held by another hand. Alternatively, one hand holding a block could "stack" it onto a block being held by another.

- *the initial state description was invalid.* In particular, if a hand was holding two blocks in the initial state, then it could stack the one onto the other, leaving the resulting stack floating in mid-air.

The first of these problems could be reproduced by performing POI for the one-block, two-hands, one-table blocks world. The resulting state-transition network in Figure 98 confirms that one hand may indeed snatch a block from another hand. The corresponding operator is shown in Figure 99. POI cannot be used for reproducing the second problem.

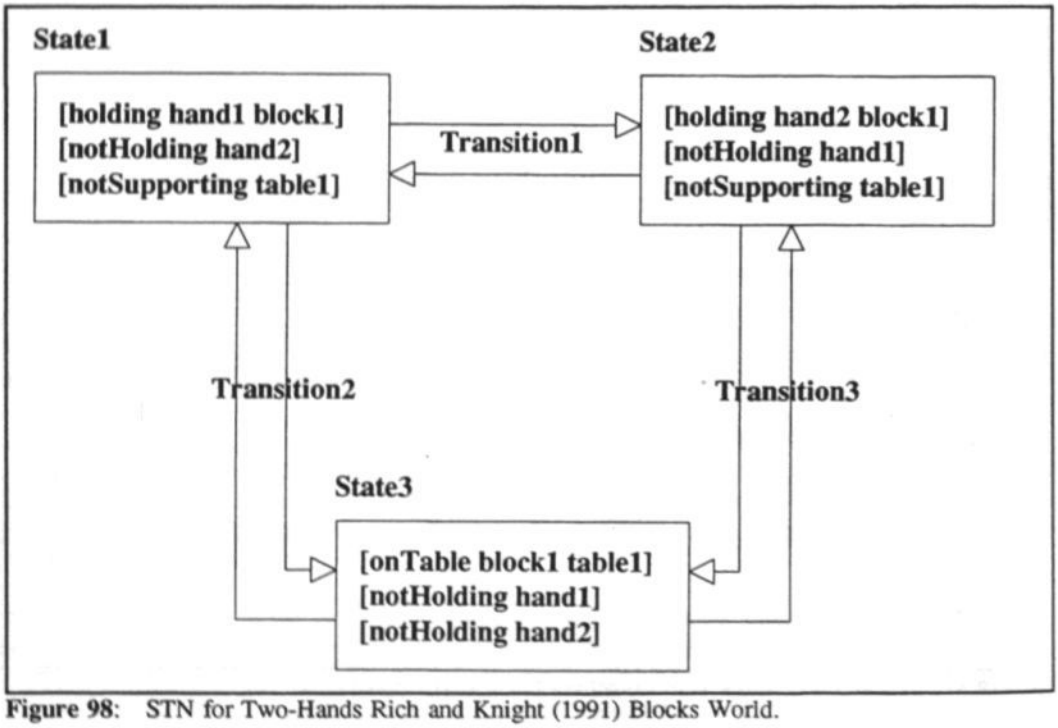

Figure 98: STN for Two-Hands Rich and Knight (1991) Blocks World.

```
EntityManager1-operator1 is:
   - my (filter) preconditions are:
      [isBlock ?Block1]
      [isTable ?Table1]
      [isHand ?Hand1]
      [isHand ?Hand2]
      [notBeneath ?Block1]
      [notOn ?Block1]
      [notOnTable ?Block1]
      [notSupporting ?Table1]
   - my delete-list is:
      [holding ?Hand1 ?Block1]
      [notHolding ?Hand2]
   - my add-list is:
      [holding ?Hand2 ?Block1]
      [notHolding ?Hand1]
```

Figure 99: Snatch Operator for Rich and Knight (1991) Blocks World.

4.8 VARYING META-HEURISTIC

Experiment P6 consisted of a series of runs for the four meta-heuristics[50]. The most interesting effects were observed for the Tank-Farm domain, rather than for the Piano-Playing domain.

Figure 100 shows how the state-transition network for the two-`Tank`, one-`Reactor` version of the Tank-Farm domain changes with varying meta-heuristic. Only the transitions change, as follows:

- *Single-Actor/Single-State-Change (SA/SSC).* With the SA/SSC meta-heuristic, transitions 1 to 4 are identified. This gives a state-transition network in which one state in which fluid is flowing can only be reached from another via the "safe" state in which no fluids are flowing (i.e., State5). This behaviour corresponds to the safety rule applied to hazardous processes by a large multinational in the petrochemical industry. The hazardous-process safety rule stipulates that only one valve may be open at a time, and that the currently-open valve must be closed fully before another can be opened. The Nilsson (1980) variant of the Blocks World exhibits similar behaviour, albeit in embryo, in that many blocks world plans must pass through a state in which all the blocks are located on the table. This state can be regarded as "gravitationally safer" than having one or more blocks held by hands or stacked on other blocks.

- *Multi-Actor/Single-State-Change (MA/SSC).* With the MA/SSC meta-heuristic, transitions 1 to 9 are identified. This gives a state-transition network where, in addition to passing through the "safe" state, one state in which fluid is flowing can be reached directly from another, with one exception. The exception is that it is not permitted to switch a reactor from loading one tank to loading another. This behaviour corresponds to the safety rule applied by the same multinational to non-hazardous processes. The non-hazardous-process safety rule also stipulates that only one valve may be open at a time, but permits one valve to be opened at the same time as another is being closed, so long as the valves do not load tanks from the same reactor.

[50] The meta-heuristics were introduced in Section 1.2.2.

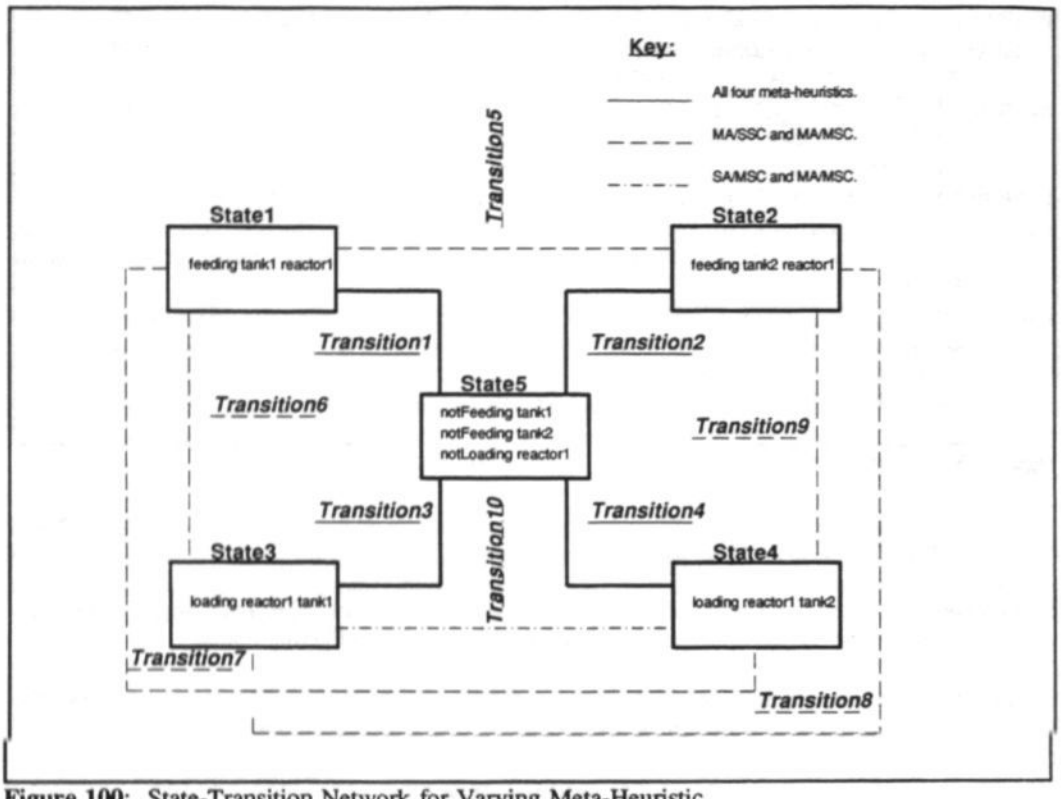

Figure 100: State-Transition Network for Varying Meta-Heuristic.

- *Single-Actor/Multi-State-Change (MA/MSC).* With the SA/MSC meta-heuristic, transitions 1 to 4 and 10 are identified. This gives a state-transition network where, in addition to passing through the "safe" state, it is permitted to switch from loading one tank from a reactor to loading another tank from the same reactor. Only one valve may be open at a time, but it is permissible to open one loading valve at the same time as another loading valve is being closed.

- *Multi-Actor/Multi-State-Change (MA/MSC).* With the MA/MSC meta-heuristic, all 10 transitions are identified. Only one valve may be open at a time, but any valve may be opened at the same time as another is being closed.

4.9 PARTITIONING VERSION-SPACE

Sections 4.4 and 4.5 have shown empirically that the complexity of the POI algorithm, as determined theoretically in Section 3.6.1, quickly reaches a point at which available memory is exhausted. It has been necessary to devise countermeasures to the combinatorial explosion. In this section, I describe a countermeasure based on the *divide-and-conquer* strategy.

This strategy is applied by partitioning the version space by the set of object-classes or object-instances in the domain. Each sub-version-space is induced separately, and the resulting set of states are merged in a process analogous to Hirsh's (1989) incremental version-space merging method. Where the version space is partitioned by object-classes, this process will be known as *class-state merging*, and where

it is partitioned by object-instances, it will be known as *instance-state merging*.

The DUC-ASS program has been enhanced to incorporate class- and instance-state merging as user-selectable alternatives to global-state induction. Comparative runs were done for the Piano-Playing domain and for the Nilsson (1980) variant of the Blocks World. Multiple runs were done, varying the numbers of instances per object-class. The results for the one-hand, two-blocks, one-table Blocks World are typical; see Figure 101.

Process	Nodes induced	Memory usage (KB)	Duration (secs)
Global induction	2624	2623.5	1697.0
Class-state merging	969	890.0	246.1
Instance-state merging	37	164.4	16.3

Figure 101: Comparison of Version-Space Partitioning for 2-Blocks World.

Class-state merging was found to be essential for performing POI for the Aircraft Scheduling and HPCE domains, as well as for the POI meta-domain. Instance-state merging would be essential for any domain with a large number of instances for one or more object-classes, e.g., a full-size Piano-Playing domain with 88 keys and 10 fingers. To date, DUC-ASS runs have been done using instance-state merging for domains with up to five instances per object-class.

4.10 INTRODUCING INHERITANCE

4.10.1 Inheritance as Countermeasure

A second countermeasure is to introduce an inheritance hierarchy of object-classes. Inheritance is beneficial primarily in that it enables the number of relationship-classes (and therefore relationship-instances) to be reduced. Since the size of the version-space lattice is related to the number of relationship-instances, introducing inheritance can reduce memory usage and runtime. The penalty is an increase in the number of object-classes needed to model a given domain.

A secondary benefit of introducing inheritance is that the number of relationship-classes per object-class is reduced. In domains with triple (or higher) constraints, the inheritance hierarchy can be carefully chosen so that no object-class is linked to more than two relationship-classes. This ensures that all domain constraints can be expressed as binary constraints, enabling the version space to be pruned to the maximum possible extent.

The desirability of ensuring that no object-class is linked to more than two relationship-classes may also explain the observation (in Section 4.1.4) that, in the Blocks World variants, there is a linear association between the numbers of object-classes and relationship-classes. In modelling a domain, there is generally some leeway in judging how many object-classes to represent. A minimum requirement is that each additional object-class must be linked to at least one relationship-class, in order to become part of the domain model. At the other extreme, each additional object-class should not be linked to more than two relationship-classes, in order to avoid the risk of being unable to model

domain constraints of arity greater than two[51]. Consequently, the numbers of object- and relationship-classes are proportional to one another, with a constant of proportionality between 1 and 2.

4.10.2 Exemplifying Benefits using Blocks World

Both benefits can be illustrated using the Nilsson (1980) variant of the Blocks World. The Blocks World entity-relationship model shown in Chapter 2 was without inheritance; see Figure 102(a). In this model of the Blocks World, blocks can assume four possible roles:

- the actee of the `holding` relationship with a hand.
- the actor of the `on` relationship with another block.
- the actee of the `on` relationship with another block.
- the actor of the `onTable` relationship with a table.

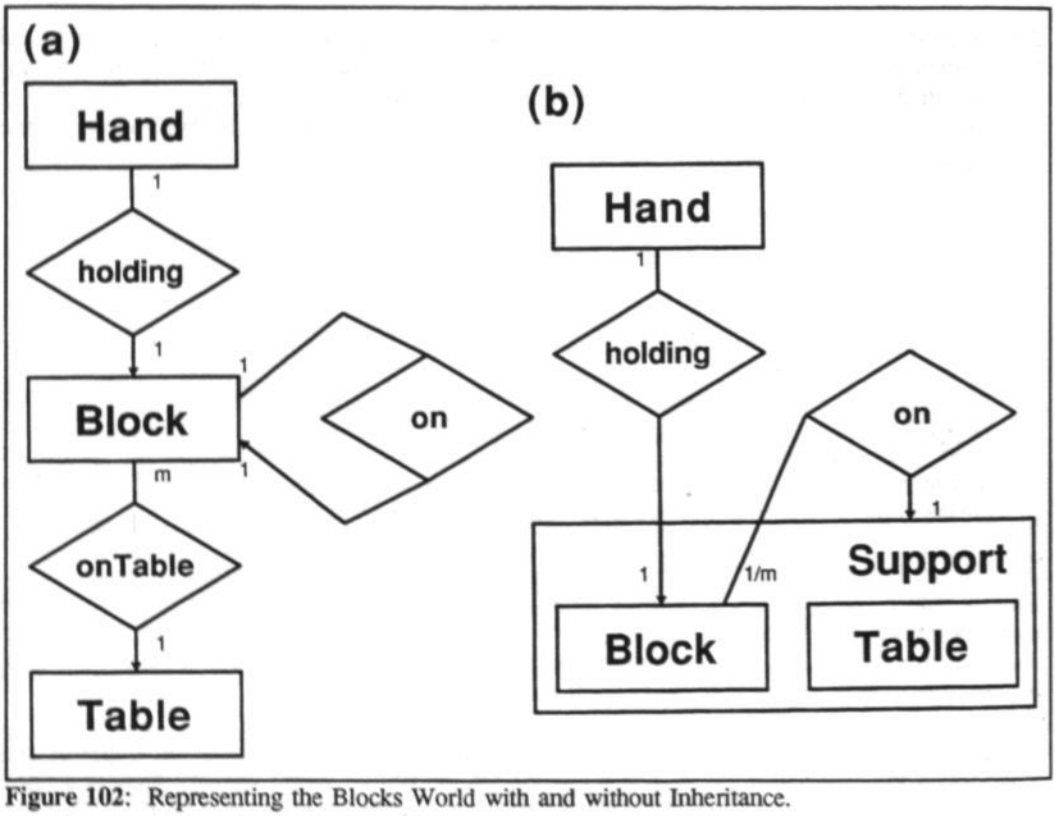

Figure 102: Representing the Blocks World with and without Inheritance.

In principle, there could be a Blocks World constraint that involved all four roles. In practice, such

[51] This leads to the following design heuristic: where an object-class is linked to more than two relationship-classes, then the domain model may be insufficiently refined.

a constraint does not exist, but, as noted in Section 3.5, the following triple constraint holds for those blocks worlds under the influence of non-zero gravity:

```
IF [notHeld ?Block1]
AND [notOn ?Block1]
AND [notOnTable ?Block1 ?Table1]
THEN INVALID.
```

This constraint requires that blocks must be either held, or on another block, or on a table.

Inheritance can be introduced by observing the similarity in the `on` and `onTable` relationship-classes[52]. Rather than concentrating on the block being stacked or unstacked, as is usual in Blocks World planning, we concentrate on the object *beneath* it. Whether that object be a block or a table, it is supporting the block being stacked/unstacked. Therefore, we identify an abstract object-class, to be named `Support`, that provides this block-supporting capability in the form of a generalised `on` relationship-class between `Block` and `Support` object-classes. The existing `Block` and `Table` object-classes inherit from this `Support` object-class, as shown in Figure 102(b). The `Block` object-class specialises `Support`, e.g., in having the `holding` relationship-class with the `Hand` object-class.

The introduction of inheritance has reduced the number of relationship-classes to two, at the penalty of introducing an additional object-class. Since there is now no object-class that can participate in more than two roles, all domain constraints are guaranteed to have an arity of 2 or less. In particular, the example triple constraint can now be expressed as the following binary constraint:

```
IF [notHeld ?Block1]
AND [notOn ?Block1]
THEN INVALID.
```

This binary constraint requires that blocks must be either held or on a support, where a support can be another block or a table.

4.10.3 Results of DUC-ASS Runs

The DUC-ASS program has been enhanced to incorporate object-class inheritance, both in global induction and in class-state merging, but not in instance-state merging. Comparative runs were done with and without inheritance for the Nilsson (1980) variant of the Blocks World, varying the numbers of `Block` object-instances. The results for the one-hand, two-blocks, one-table Blocks World using global induction are shown in Figure 103.

Process	Nodes induced	Memory usage (KB)	Duration (secs)
Without inheritance	2624	2623.5	1697.0
With inheritance	528	356.4	65.3

Figure 103: Comparison of 2-Block World with and without Inheritance.

[52] Not least in their names. Note, too, that some AI references elide the two relationship-classes into one relationship-class, e.g., Bundy, Burstall, Weir and Young (1980) and Charniak and McDermott (1985).

Inspection of the output shows that the binary constraint has indeed been effective. Figure 104 shows that "floating blocks" have been eliminated from the states induced for the one-hand, one-block, one-table Blocks World.

```
POI Step (2.3): EntityManager1 identifying states ...

... EntityManager1-StateInstance 1 created from EntityManager1-Node 37:
      [holding hand1 block1]
      [notBeneath block1]
      [notBeneath table1]
      [notOn block1]

... EntityManager1-StateInstance 2 created from EntityManager1-Node 40:
      [notBeneath block1]
      [notHeld block1]
      [notHolding hand1]
      [on block1 table1]

POI Step (2.3) completed; EntityManager1 identified states.
```

Figure 104: States Induced for One-Block World with Inheritance.

4.10.4 Comparing Effectiveness of Countermeasures

Finally, the effectiveness of inheritance and version-space partitioning as countermeasures to the combinatorial explosion were compared. The results for the one-hand, two-blocks, one-table Blocks World with and without inheritance and using global induction and class-state merging are summarised in Figure 105. These results show that inheritance is more effective than class-state merging, but that the combination is better than either countermeasure alone.

Process	Nodes induced	Memory usage (KB)	Duration (secs)
Global induction, without inheritance	2624	2623.5	1697.0
Global induction, with inheritance	528	356.4	65.3
Class-state merging, without inheritance	969	890.0	246.1
Class-state merging, with inheritance	229	178.2	31.5

Figure 105: Comparing Effectiveness of Countermeasures.

4.11 POI META-DOMAIN

Chapter 3 noted that the POI ontology is a meta-representation. In other words, the POI ontology supports the definition of domains using the object-relationship model, enhanced to represent domain constraints as binary exclusion-rules. At the same time, the POI ontology has itself been documented as an object-relationship model; domain constraints are expressed in the description of the POI algorithm. Other examples of meta-representations are to be found in Gruber (1992) and in Pedersen (1995).

The fact that the POI ontology is a meta-representation can be exploited. It suggests that it would be possible to perform DUC-ASS runs with the POI ontology modelled as "just another" domain, i.e., as a *meta-domain*. A DUC-ASS run for the POI meta-domain will be known as a *meta-run*. The results of a meta-run should be a set of planning operators which includes one or more of the steps of the POI algorithm.

The POI ontology has been successfully modelled using the DUC-ASS facilities. With one instance per object-class, the theoretical maximum size of the version space is 2^{73} nodes. This is considerably larger than the HPCE domain's maximum version-space size. As combining class-state merging, inheritance, and one instance per object-class were essential for HPCE runs, the same combination was also essential for POI meta-runs.

Experimentation showed that it was not possible to achieve a full meta-run. The DUC-ASS program ran out of memory during version-space construction, i.e., during Step (2.2). Inspection of the session trace showed that all the class-states had been induced, and that the subsequent merging process was well advanced before memory was exhausted. Repeating the meta-run on PCs with more memory showed that the merging process would have needed more than the 16 MB memory addressable in the protected mode. Despite substantial refinement of the POI meta-domain and optimisation of the DUC-ASS's use of memory, memory usage could not be reduced below 16 MB. Therefore, the attempt to achieve a full meta-run had to be abandoned.

Although a full meta-run could not be completed, some conclusions could be drawn, as follows:

- The failure was attributable to a limitation in the implementation environment, and not to a limitation in the POI ontology or algorithm.

- Inspection of the induced class-states showed that, if the 16 MB limitation had been absent, a set of planning operators would have been obtained.

- The extent to which the merging process had progressed showed that a moderate increase in addressable memory would have enabled a full meta-run to be completed. The memory required was estimated at not more than 32 MB.

Even if a full meta-run had been completed, the restrictions on the POI meta-domain model would have made it impossible to induce a state-transition network representing the *complete* POI algorithm. The same situation arose for the HPCE runs. The minimum number of object-instances needed for the complete POI algorithm to be induced would have had to include:

- two EntityInstances,

- three RelationInstances,

- two ExclusionRules,
- two StateInstances,
- two TransitionInstances, and
- two PlanningOperators.

4.12 CHAPTER 4 CONCLUSIONS

The purpose of this chapter was to describe a programme of open-loop experiments designed to demonstrate that the POI algorithm was capable of inducing planning operators and to investigate the algorithm's complexity. Both programme goals have been achieved.

The POI algorithm, as implemented in the DUC-ASS program, has induced sets of planning operators for eight domains ranging from one to 24 object-classes. Extracts of the outputs from three selected domains have been documented in this chapter. Given a suitable domain model, planning-operator sets to be found in the AI literature can be reproduced. Furthermore, novel planning operators can be induced, as shown using the HPCE domain. These novel planning operators include several examples that could not be generated by other operator-learning algorithms.

The sensitivity of the POI algorithm has been investigated by varying object-classes, object-instances, relationship-classes, domain constraints, and the meta-heuristic used to extract transitions. These investigations have been summarised in this chapter.

The POI algorithm is combinatorially explosive. The complexity of the algorithm has been investigated empirically, both within a given domain and across several domains. Within-domain complexity is consistent with exponential behaviour, and across-domain complexity is consistent with a third-order polynomial.

The effectiveness of the countermeasures was evaluated against the baseline algorithm. Comparative runs with and without countermeasures show that, for small numbers of object-classes and small numbers of object-instances per class, introducing inheritance is more effective than version-space partitioning. Combining both countermeasures is more effective than either countermeasure alone. The introduction of inheritance has also been shown to be effective in enabling domain constraints to be modelled completely using binary exclusion-rules.

Finally, the fact that the POI ontology is a meta-representation has been exploited to perform DUC-ASS runs for the POI meta-domain. Limitations attributable to the implementation environment caused these meta-runs to be aborted due to insufficient memory. However, this occurred after all the class-states had been induced. Inspection of the induced class-states showed that, given a doubling in addressable memory, the meta-runs could have been completed successfully.

5 CLOSED-LOOP EXPERIMENTS

The purpose of this chapter is to describe a series of experiments with the multi-agent application of the Planning Operator Induction (POI) algorithm. The experiment series was aimed at closed-loop testing of the algorithm. Closed-loop testing was effected by embedding the POI algorithm in an agent (see Figure 106). The resulting *POIAgent* was given a series of goals to achieve by interacting with other agents in its environment. The series of goals was so designed that the POIAgent acquired knowledge about the other agents, induced a set of planning operators from the acquired knowledge, generated a plan using the induced operators, and executed that plan successfully.

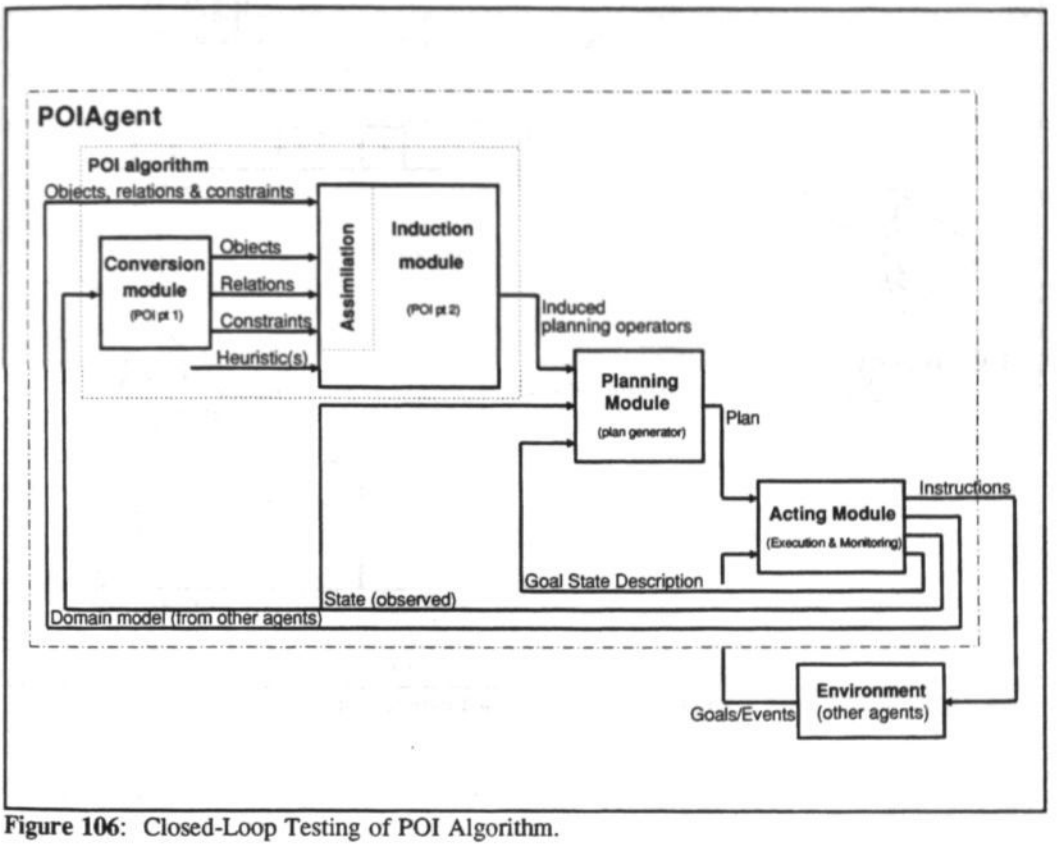

Figure 106: Closed-Loop Testing of POI Algorithm.

The planning and acting modules and the other agents shown in Figure 106 must be regarded as the "test harness" for the POI algorithm. The purpose of the test harness is to close the loop from the POI algorithm's outputs to its inputs. Details of the test harness are unimportant to this thesis; more details may be found in (Grant, 1991) and (Grant, 1992b). To couple the input of the induction process (POI Part 2) to the test harness, POI Part 1 had to be implemented.

Two groups of experiments were done. The first group demonstrated single-agent learning in a multi-agent environment, as outlined above. The second group demonstrated multi-agent learning, in that two POIAgents were each given an individual series of goals to achieve, and then had to pool their

knowledge in order to induce a complete and correct set of planning operators[53].

This chapter contains six sections. Section 5.1 describes the design of the experiments. Section 5.2 documents the predictions. Section 5.3 compares the predictions with one another, and Section 5.4 compares the experiment results with the predictions. Section 5.5 demonstrates closing the loop, i.e., causing an agent to induce a set of planning operators, to generate a plan using the induced operator-set, and to executed the generated plan. Section 5.6 summarises the chapter, highlighting the chapter's research contributions.

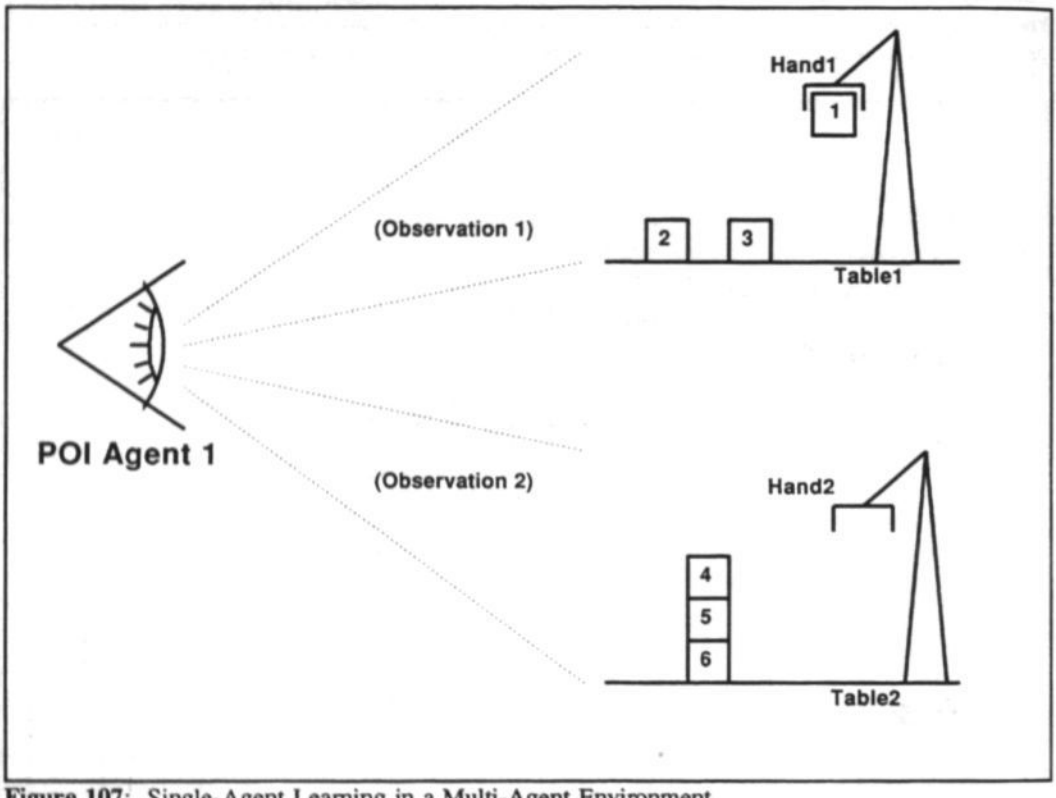

Figure 107: Single-Agent Learning in a Multi-Agent Environment.

5.1 CLOSED-LOOP EXPERIMENT DESIGN

5.1.1 Single- and Multi-Agent Learning Experiments

The single- and multi-agent learning experiments are illustrated in Figure 107 and Figure 108. Figure 107 depicts the experimental set-up for single-agent learning in a multi-agent environment. A single POIAgent is shown observing two separate blocks worlds. Each object in the blocks worlds is modelled as a (non-intentional) agent. From the POIAgent's point-of-view, the blocks world objects are the "other agents" in its environment. It must acquire information about the objects, the inter-object

[53] Complete and correct by comparison with an oracle.

relationships, and the constraints on these relationships from each observation. Before induction can begin, the POIAgent must merge and generalise the acquired information.

Figure 108 depicts the experimental set-up for multi-agent learning in a multi-agent environment. Two POIAgents are shown, each observing a separate blocks world. As in the single-agent learning experiments, each blocks world object is modelled as a non-intentional agent. From each POIAgent's point-of-view, the objects in its blocks world are the "other agents" in its environment. It must acquire information about the objects, the inter-object relationships, and the constraints on these relationships from each observation. The observations are so chosen that neither agent has acquired sufficient information to induce a complete and correct set of planning operators. Before planning operator induction can begin, one POIAgent (the *sender*) must pass its acquired information to the other (the *recipient*), so that the latter can merge and generalise the pooled information. Only at the time of exchanging information do the POIAgents become aware of each other's existence. At no time does one POIAgent become aware of the existence of the other's blocks world. Although this is not a requirement for testing, it has been done to demonstrate that the situation in Figure 107 does not apply.

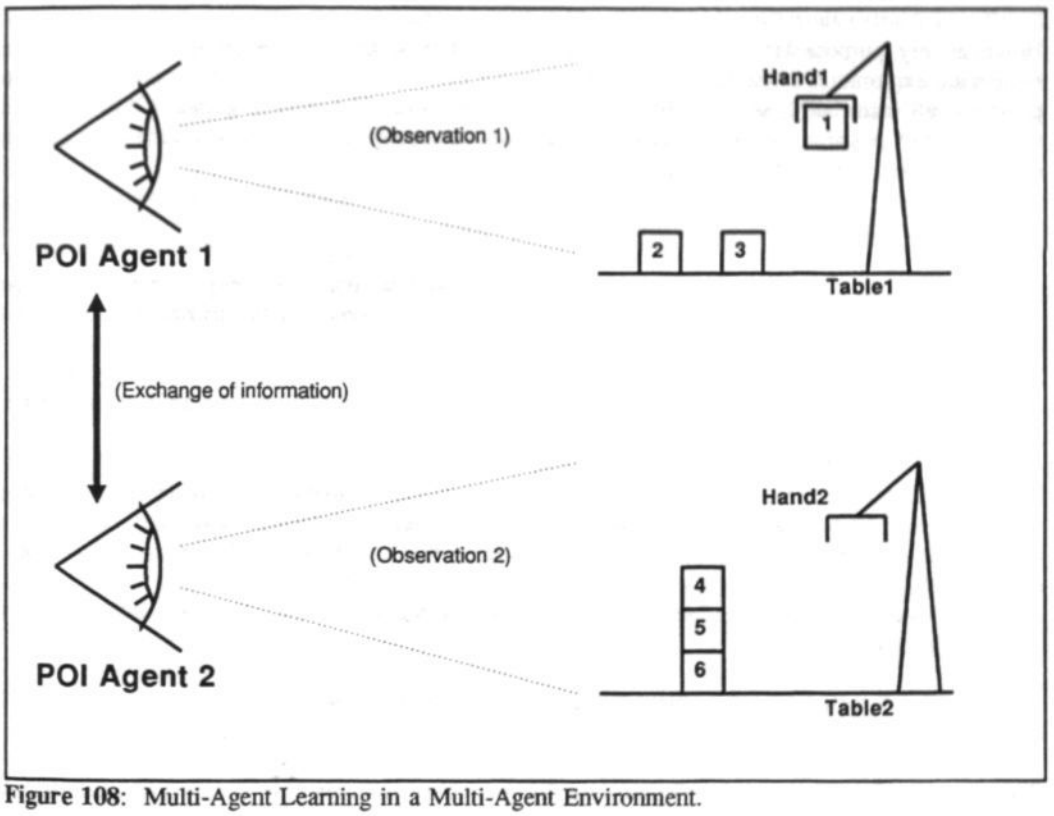

Figure 108: Multi-Agent Learning in a Multi-Agent Environment.

5.1.2 Purpose of Experimentation

The purpose of the closed-loop testing is solely to demonstrate that the POI algorithm behaves as expected when the loop from outputs to inputs is closed. In particular, the experiments show that the learning agent gains novel knowledge from unsequenced observations of world states.

I emphasise that the observations can be unsequenced. Many other authors (e.g., Vere (1978), Porter and Kibler (1986), Kadie (1988), Muggleton (1990), and Alterman and Zito-Wolf (1993)) have demonstrated the learning of domain knowledge for planning purposes by inducing planning-operator descriptions. However, in every case they have had to use situation-action pairs - called "before-and-after pairs" by Vere - and/or situation sequences as the input to induction. The sequencing information in their inputs has been crucial to their induction algorithms.

This thesis does not claim that the POI algorithm guarantees to induce a complete set of planning operators for a given domain from any set of world-state observations. Indeed, as the experiments in this chapter show, the POI algorithm could be employed as an intelligent tutoring tool to investigate how an agent's "knowledge" of a domain develops as it acquires observations[54]. At an intermediate stage during the knowledge-gaining process, the agent would induce incomplete or inaccurate operator-sets. These operator-sets could be used to generate plans, which, when executed, would fail in characteristic ways. These failure characteristics could be indexed by the information which the agent lacked. A student's behaviour could then be diagnosed by matching its characteristics against the index of characteristics to determine what domain information the student was lacking.

However, my purpose is not to develop an intelligent tutoring tool. To do so would require a systematic exploration of the failure characteristics for every possible set of domain observations. In principle, all factorial(N) sets of world-state observations would have to be explored[55], where N is the number of possible world states. For my purposes, it is sufficient that I can find at least one set of observations with the following properties:

- The observations are domain state descriptions.
- The set contains at least two observations, both so that single-agent learning can be demonstrated incrementally, and so that the observations can be distributed across two POIAgents to demonstrate multi-agent learning.
- The observations do not form a sequence, i.e, no observation is related to any other observation by a shared transition.
- The set of observations contains sufficient information to induce a complete set of planning operators for the domain. The sufficiency of the contained information will be judged on the basis of the objects, relationships and constraints that can be extracted from the observations.
- There is a proper subset of the observations for which the contained information is incomplete.
- The domain is one for which an oracle (i.e., a non-POI source of sets of planning operators) exists.

The demonstration of other properties, such as the minimality or uniqueness of the set of observations,

---

[54] I put "knowledge" in quotes here because, from the global viewpoint, it is justified, but not a true belief. From the agent's viewpoint, its beliefs are both true and justified. In an application where no global viewpoint exists, there would be no basis to distinguish knowledge from "knowledge".

[55] In practice, this number could be reduced drastically, without resorting to the Single Fault Assumption, by exploiting symmetries in the domain. Readers interested in applying the POI algorithm to intelligent tutoring will find many relevant pointers in this chapter.

is specifically excluded. However, it is convenient to find:

- A domain whose complexity is as low as possible, and
- A set of observations whose cardinality is as low as possible, subject to the minimum of two observations,

because these properties reduce the number and complexity of the test runs needed.

5.1.3 Basis for Validation

Experiment validation has been done by *Hypothesis Validation*, i.e., by comparing the succession of variable-values identified by the POIAgent when the observations are presented incrementally against the predicted succession of variable-values. The variables produced by the POI algorithm include:

- The object-classes, object-instances, relationship-classes and constraints extracted by Part 1 of the POI algorithm ("Acquisition") from the observed world-states.
- The planning operators induced by Part 2 of the POI algorithm ("Induction") from these object-classes, object-instances, relationship-classes and constraints.
- Other intermediate variables, notably state- and transition-instances.

In the absence of an independent, interactive oracle for the blocks world, the induced planning operators cannot be predicted for all blocks world variants. In particular, planning operators based on incomplete or incorrect knowledge of the blocks world have not been reported in the literature. Therefore, hypothesis validation has been done by comparing the objects, relationships, and constraints identified by the POIAgent, supplemented where necessary by comparing other intermediate variables.

5.1.4 Validation Procedure for Single-Agent Learning

The hypothesis validation procedure for single-agent learning is as follows:

(1) Start with a *naive* POIAgent, i.e., one that had no knowledge whatsoever of the problem domain.

(2) Present one of the observations to the POIAgent.

(3) Note the object-classes, object-instances, relationship-classes and constraints identified by the POIAgent.

(4) If the set of observations is not exhausted, go back to step (2).

(5) Instruct the POIAgent to achieve a goal that would force it to generate a plan which, on execution, would pass through one or more novel states.

(6) Note the planning operators induced by the POIAgent.

(7) Note the plan generated by the POIAgent.

(8) Note whether the POIAgent successfully executed the plan by causing the problem domain to pass through at least one state novel to the POIAgent.

This procedure was repeated, varying the order of presentation of the set of observations. The results of the repetitions were compared in order to:

- Confirm that the induced planning operators were independent of the order of presentation.
- Show that the object-classes, object-instances, relationship-classes, and constraints identified by the POIAgent were order-dependent.

The results for the single-agent learning experiments are presented in Section 5.5.1.

5.1.5 Validation Procedure for Multi-Agent Learning

The hypothesis validation procedure for multi-agent learning is as follows:

(1) Start with two naive POIAgents.

(2) Present one proper subset of the observations to one POIAgent and the set of the remaining observations to the other POIAgent. Neither set must be empty.

(3) Note the object-classes, object-instances, relationship-classes and constraints identified by each POIAgent. Show that neither POIAgent had sufficient information to induce a complete set of planning operators.

(4) Cause one POIAgent (the *sender*) to transmit its lists of object-classes, relationship-classes and constraints to the other POIAgent (the *recipient*).

(5) Instruct the recipient POIAgent to achieve a goal that would force it to generate a plan which, on execution, would pass through one or more novel states.

(6) Note the planning operators induced by the recipient POIAgent.

(7) Note the plan generated by the recipient POIAgent.

(8) Note whether the recipient POIAgent successfully executed the plan by causing the problem domain to pass through at least one state novel to both POIAgents.

This procedure was repeated, exchanging the sender/recipient roles of the POIAgents. The results of the repetitions were compared in order to confirm that the induced planning operators were independent of the direction of transmission of information.

The results for the multi-agent learning experiments are presented in Section 5.5.2.

5.1.6 Choice of Illustrative Domain

One way of verifying the completeness of the set of planning operators would be to compare them with the planning operators obtained from an oracle. In the previous chapter we have discussed the difficulty of obtaining oracles for planning operator sets. Alternatively, verification could be done by using a domain for which all valid states are known. The agent would then be tasked with generating plans for all possible pairs of valid states. The correctness of the plans would then be checked by executing them.

This thesis adopts a combination of these ways of verifying the induced planning operators. This is done by showing that a POIAgent can:

- induce a complete set of planning operators after all the observations have been presented. Completeness is verified by comparing the induced operator-set against the oracle's operator-set.

- generate a correct plan which passes through at least one *novel* world state, i.e., a state which has neither been previously observed nor is a renaming of a previously observed state.

- execute the generated plan successfully.

The blocks world has been chosen. More specifically, the Nilsson's (1980) blocks world with three blocks, one hand and one table - the *baseline variant* - has been chosen because:

- Nilsson (1980, p.281) documents a set of planning operators for the blocks world.

- Nilsson (1980, Figure 7.3, p. 283) provides a figure depicting the complete state-space for the 3-blocks, 1-hand, 1-table variant of the blocks world.

- sensitivity analysis of this blocks world variant has been done by a large number of authors.

For consistency in describing the closed-loop experiments, I adopt Nilsson's (1980) terminology for blocks world relationships.

5.1.7 Choice of Set of Observations

This section describes how the set of observations was chosen.

There are 22 valid states in Nilsson (1980)'s depiction of the state space for the 3-blocks, 1-hand, 1-table blocks world. In the worst case, it might be necessary to present a set of observations containing all 22 valid states in order for an induction algorithm to learn a complete set of four operators. In practice, a very much smaller set suffices, because the POI algorithm generalises the states it observes. As noted in Chapter 2, Nilsson's (1980) state-space can be generalised to just five state-classes, corresponding to the five rows of states in Nilsson's figure.

For convenience, we wish to present as small a set of observations as possible. The set must in any case contain four or less state descriptions in order to guarantee that at least one novel state exists. The set cannot contain just one state description because there would then be insufficient observations either to investigate knowledge-gaining or to distribute over multiple POIAgents. More importantly, the POIAgent would observe the hand either in the primary `holding` relationship or in the inverse

handempty relationship, but not in both. It must observe both in order to identify the constraint:

```
IF handempty ?Hand1 AND holding ?Hand1 ?Block1 THEN INVALID.
```

This argument may be generalised to encompass the other relationships. To observe both the primary and inverse relationships, the set of observations must contain at least two state descriptions. However, the observations must not be adjacent to avoid implying that the set of observations constitutes a sequence (i.e., represents a plan segment).

5.1.8 Observations Identified

A set of two observations has been identified for the 3-blocks, 1-hand, 1-table world which meets these requirements and provides sufficient information for the POI algorithm to induce a complete set of planning operators. The set of observations is given in Figure 109.

Observation (1):

```
clear block2
clear block3
holding hand1 block1
ontable block2 table1
ontable block3 table1
```

Observation (2):

```
clear block4
on block4 block5
on block5 block6
ontable block6 table2
handempty hand2
```

Figure 109: Set of Observations Meeting Requirements.

In isolation, Observation (1) shows that:

- There are three object-classes:
  - Hand, with one instance: `hand1`.
  - Block, with three instances: `block1`, `block2` and `block3`.
  - Table, with one instance: `table1`.
- Hands can have a `holding` relationship with blocks.
- Blocks can have an `ontable` relationship with tables.
- Blocks can be `clear`.
- A block that is `ontable` can also be `clear`. Any constraint generated by default by the POI algorithm to exclude such situations will be marked UNUSEABLE.
- Several blocks can be on the same table simultaneously. Any constraint generated by default by the POI algorithm to exclude such situations will be marked UNUSEABLE.

In isolation, Observation (2) shows that:

- There are three object-classes:
  - Hand, with one instance: `hand2`.
  - Block, with three instances: `block4`, `block5` and `block6`.
  - Table, with one instance: `table2`.
- Hands can be `handempty`.
- Blocks can have an `on` relationship with other blocks.
- Blocks can have an `ontable` relationship with tables.
- Blocks can be `clear`.
- A block that is `on` another block can also be `clear`. Any constraint generated by default by the POI algorithm to exclude such situations will be marked UNUSEABLE.
- A block with another block `on` it can be `on` a third block. Any constraint generated by default by the POI algorithm to exclude such situations will be marked UNUSEABLE.
- A block `ontable` a table can have another block `on` it. Any constraint generated by default by the POI algorithm to exclude such situations will be marked UNUSEABLE.

Taken together, Observations (1) and (2) show that:

- There are three object-classes:
  - Hand, with two instances: `hand1` and `hand2`.
  - Block, with six instances: `block1`, `block2`, `block3`, `block4`, `block5`, and `block6`.
  - Table, with two instances: `table1` and `table2`.
- Hands can have a `holding` relationship with blocks.
- Hands can be `handempty`. By implication, this is the inverse of `holding`.
- Blocks can have an `on` relationship with other blocks.
- Blocks can have an `ontable` relationship with tables.
- Blocks can be `clear`. By implication, this is the inverse of `on`.
- A block that is `on` another block can also be `clear`. Any constraint generated by default by the POI algorithm to exclude such situations will be marked UNUSEABLE.
- A block that is `ontable` can also be `clear`. Any constraint generated by default by the POI algorithm to exclude such situations will be marked UNUSEABLE.
- Several blocks can be on the same table simultaneously. Any constraint generated by default by the POI algorithm to exclude such situations will be marked UNUSEABLE.

- A block with another block `on` it can be `on` a third block. Any constraint generated by default by the POI algorithm to exclude such situations will be marked UNUSEABLE.

- A block `ontable` a table can have another block `on` it. Any constraint generated by default by the POI algorithm to exclude such situations will be marked UNUSEABLE.

Since neither observation shows:

- A hand holding multiple blocks, or

- A block with multiple blocks on it, or

- A block on multiple blocks, or

- Two blocks mutually on each other, or

- A block on itself,

the POI algorithm will, by the default rule, generate constraints to exclude such situations.

5.1.9 Experiment Procedure

A given POIAgent may be presented with Observation (1), with Observation (2), or with both Observations. A POIAgent that has been presented with just Observation (1) will be denoted as a "$\text{POIAgent}_{\text{Obs(1)}}$", one that has been presented with just Observation (2) will be denoted as a "$\text{POIAgent}_{\text{Obs(2)}}$", and one that has been presented with both observations will be denoted as a "$\text{POIAgent}_{\text{Obs(1\&2)}}$".

First, predictions are made as to the objects, relationships and constraints that the POIAgent would identify, i.e., the "knowledge" possessed by an $\text{POIAgent}_{\text{Obs(1)}}$, an $\text{POIAgent}_{\text{Obs(2)}}$, and an $\text{POIAgent}_{\text{Obs(1\&2)}}$.

Second, these predictions are compared, as follows:

- The "knowledge" possessed by a $\text{POIAgent}_{\text{Obs(1)}}$ is compared with the "knowledge" possessed by a $\text{POIAgent}_{\text{Obs(2)}}$ to identify what information is lacking in each observation, and whether this lack of knowledge is dependent on the order in which the observations are presented.

- The sum of the "knowledge" possessed by a $\text{POIAgent}_{\text{Obs(1)}}$ and by a $\text{POIAgent}_{\text{Obs(2)}}$ is compared with the knowledge possessed by a $\text{POIAgent}_{\text{Obs(1\&2)}}$.

Third, the experiments are run. Figure 110 shows the experiment design for demonstrating single-agent learning in a multi-agent environment. The experiment design is shown as a state-transition network. At the start, the POIAgent had no knowledge of the problem domain, and the two blocks worlds were uninitialised. The user presents the first observation by instructing the POIAgent to initialise one of the blocks worlds. After noting the objects, relationships and constraints identified by the POIAgent from the first observation, the user presents the second observation by instructing the POIAgent to initialise the other blocks world. The objects, relationships and constraints identified by the POIAgent from both observations are noted. Then the user triggers the induction process by instructing the POIAgent to change the state of one of the blocks worlds in such a way that a plan was needed.

Lacking planning operators, the POIAgent performs induction (i.e., Part 2 of the POI algorithm). The induced operator-set is used (by the Planning module) to generate a plan, which is then executed (by the Acting module)[56]. As shown by the dashed lines, the experiment is repeated for the reverse order of presentation of the two observations.

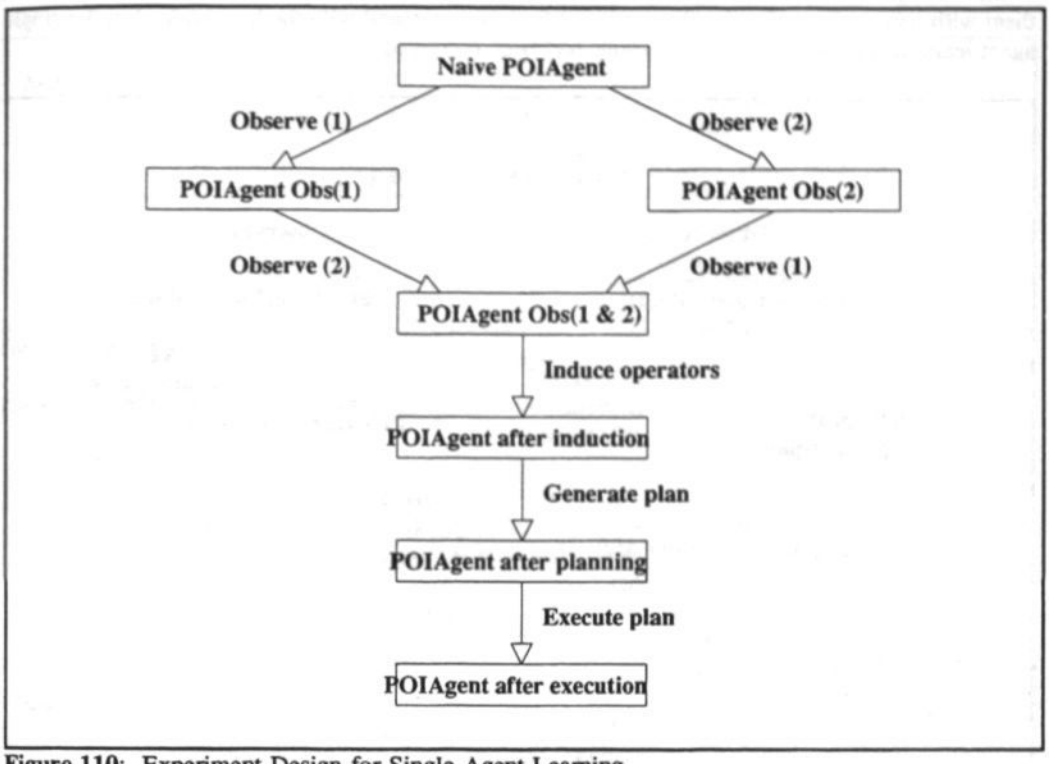

Figure 110: Experiment Design for Single-Agent Learning.

Finally, the following comparisons are made:

- Between the predicted lists of objects, relationships and constraints, as noted after Observation (1) and those noted after both Observations.
- Between the predicted lists of objects, relationships and constraints, as noted after Observation (2) and those noted after both Observations.
- Between the predicted lists of objects, relationships and constraints, as noted after Observation (1) and those noted after Observation (2).
- Between the predicted and noted lists of objects, relationships and constraints.

There was no need for additional predictions in the multi-agent learning case. Figure 111 shows the experiment design for demonstrating multi-agent learning, also shown as a state-transition network. At the start, two POIAgents were created with no knowledge of the problem domain, and the two

[56] This process is demonstrated in Section 5.3.7.

blocks worlds were uninitialised. The experiment was run by the user presenting one observation to one POIAgent and the other observation to the other POIAgent. The objects, relationships and constraints identified by each POIAgent were noted to check that the lists were identical to those noted in the single-agent learning case after the presentation of one observation. Then the user instructed one of the POIAgents to acquire the other's list of objects, relationships and constraints and to combine them with its own lists. Finally, the combined list was compared with the list obtained in the single-agent learning case after both observations had been presented.

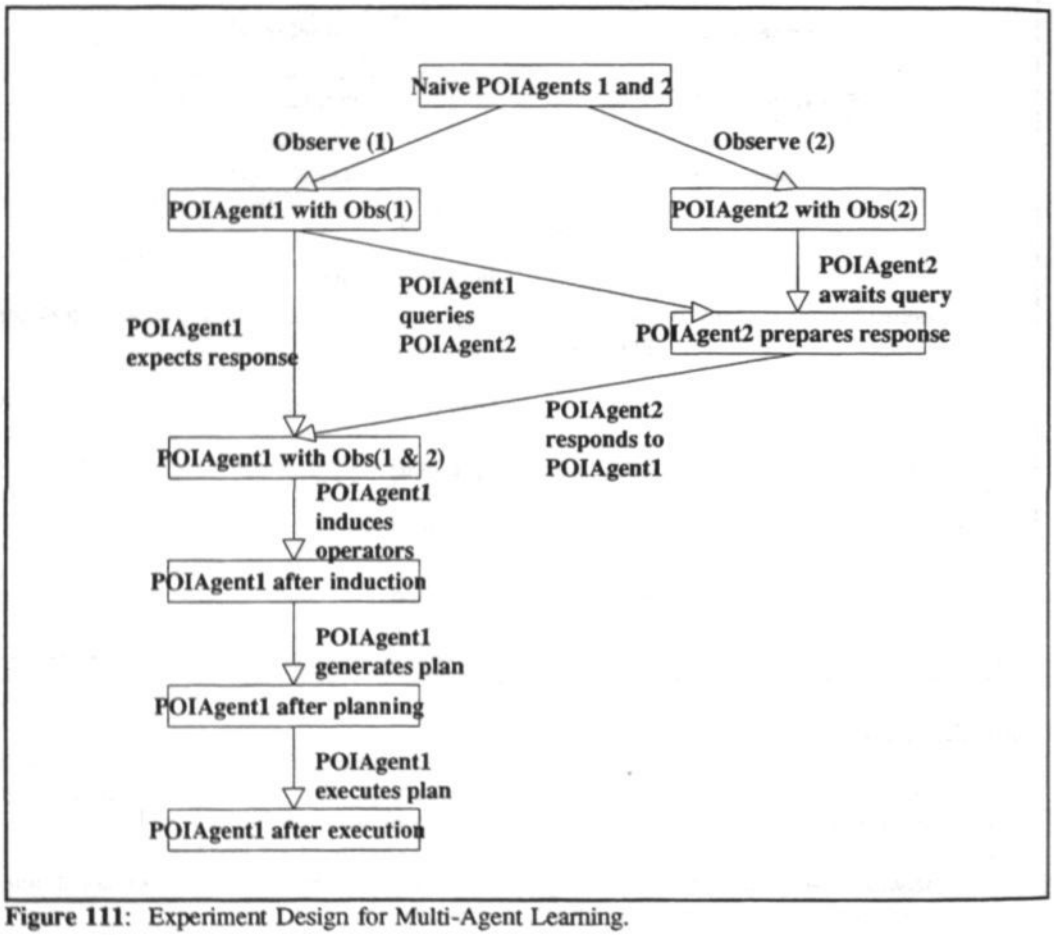

Figure 111: Experiment Design for Multi-Agent Learning.

5.2 PREDICTIONS

Predictions concentrate on the relationships and constraints, because:

- The same three object-classes are identified immediately if either Observation (1) or Observation (2) is presented to a POIAgent.
- The differences in the lists of object-instances are unimportant here.

The relationships have been predicted by manually extracting them from the state descriptions. The

constraints have been predicted by generating manually all pairs of the relationships which have a variable in common. Then the constraints are predicted as being marked UNUSEABLE if they match any subset of the state description. In some cases, there are several ways in which the pair can have a variable in common. For example, two `holding ?Hand ?Block` relationships can have either the `?Hand` variable or the `?Block` variable in common. This results in two different constraints.

Marked USEABLE:

```
IF holding ?Hand1 ?Block1 AND holding ?Hand2 ?Block1 THEN INVALID.
IF holding ?Hand1 ?Block1 AND holding ?Hand1 ?Block2 THEN INVALID.
IF holding ?Hand1 ?Block1 AND ontable ?Block1 ?Table1 THEN INVALID.
IF holding ?Hand1 ?Block1 AND clear ?Block1 THEN INVALID.
IF ontable ?Block1 ?Table1 AND ontable ?Block1 ?Table2 THEN INVALID.
```

Marked UNUSEABLE:

```
IF ontable ?Block1 ?Table1 AND ontable ?Block2 ?Table1 THEN INVALID.
IF ontable ?Block1 ?Table1 AND clear ?Block1 THEN INVALID.
```

Figure 112: Predicted Constraints for $POIAgent_{Obs(1)}$.

The predicted identifications of relationships and constraints are as follows:

- A $POIAgent_{Obs(1)}$ is predicted as knowing:
  - The list of relationships {[`holding ?Hand ?Block`], [`ontable ?Block ?Table`], [`clear ?Block`]}, and
  - The constraints listed in Figure 112.

Marked USEABLE:

```
IF on ?Block1 ?Block1 THEN INVALID.
IF on ?Block1 ?Block2 AND on ?Block1 ?Block3 THEN INVALID.
IF on ?Block1 ?Block2 AND on ?Block3 ?Block2 THEN INVALID.
IF on ?Block1 ?Block2 AND on ?Block2 ?Block1 THEN INVALID.
IF on ?Block1 ?Block2 AND ontable ?Block1 ?Table1 THEN INVALID.
IF on ?Block1 ?Block2 AND clear ?Block2 THEN INVALID.
IF ontable ?Block1 ?Table1 AND clear ?Block1 THEN INVALID.
IF ontable ?Block1 ?Table1 AND ontable ?Block2 ?Table1 THEN INVALID.
IF ontable ?Block1 ?Table1 AND ontable ?Block1 ?Table2 THEN INVALID.
```

Marked UNUSEABLE:

```
IF on ?Block1 ?Block2 AND clear ?Block1 THEN INVALID.
IF on ?Block1 ?Block2 AND on ?Block2 ?Block3 THEN INVALID.
IF on ?Block2 ?Block1 AND ontable ?Block1 ?Table1 THEN INVALID.
```

Figure 113: Predicted Constraints for $POIAgent_{Obs(2)}$.

- A $POIAgent_{Obs(2)}$ is predicted as knowing:
  - The list of relationships {[`handempty ?Hand`], [`on ?Block ?Block`], [`ontable ?Block ?Table`], [`clear ?Block`]}, and
  - The constraints listed in Figure 113.

Marked USEABLE:

```
IF holding ?Hand1 ?Block1 AND holding ?Hand2 ?Block1 THEN INVALID.
IF holding ?Hand1 ?Block1 AND holding ?Hand1 ?Block2 THEN INVALID.
IF holding ?Hand1 ?Block1 AND handempty ?Hand1 THEN INVALID.
IF holding ?Hand1 ?Block1 AND on ?Block1 ?Block2 THEN INVALID.
IF holding ?Hand1 ?Block1 AND on ?Block2 ?Block1 THEN INVALID.
IF holding ?Hand1 ?Block1 AND ontable ?Block1 ?Table1 THEN INVALID.
IF holding ?Hand1 ?Block1 AND clear ?Block1 THEN INVALID.
IF on ?Block1 ?Block1 THEN INVALID.
IF on ?Block1 ?Block2 AND on ?Block1 ?Block3 THEN INVALID.
IF on ?Block1 ?Block2 AND on ?Block3 ?Block2 THEN INVALID.
IF on ?Block1 ?Block2 AND on ?Block2 ?Block1 THEN INVALID.
IF on ?Block1 ?Block2 AND ontable ?Block1 ?Table1 THEN INVALID.
IF on ?Block1 ?Block2 AND clear ?Block2 THEN INVALID.
IF ontable ?Block1 ?Table1 AND ontable ?Block1 ?Table2 THEN INVALID.
```

Marked UNUSEABLE:

```
IF on ?Block1 ?Block2 AND clear ?Block1 THEN INVALID.
IF ontable ?Block1 ?Table1 AND clear ?Block1 THEN INVALID.
IF ontable ?Block1 ?Table1 AND ontable ?Block2 ?Table1 THEN INVALID.
IF on ?Block1 ?Block2 AND on ?Block2 ?Block3 THEN INVALID.
IF on ?Block2 ?Block1 AND ontable ?Block1 ?Table1 THEN INVALID.
```

Figure 114: Predicted Constraints for $POIAgent_{Obs(1\&2)}$.

- A $POIAgent_{Obs(1\&2)}$ is predicted as knowing:
  - A list of relationships which is the union of the lists of relationships possessed by a $POIAgent_{Obs(1)}$ and by a $POIAgent_{Obs(2)}$, namely {[`holding ?Hand ?Block`], [`handempty ?Hand`], [`on ?Block ?Block`], [`ontable ?Block ?Table`], [`clear ?Block`]}, and
  - The constraints listed in Figure 114.

5.3 COMPARING AGENTS' KNOWLEDGE

5.3.1 What One Observation Adds over the Other

Based on these predictions, it is possible to highlight what Observation (2) adds to Observation (1), and vice versa. This is best expressed as the knowledge that a POIAgent lacks before it has been presented with both observations. The knowledge that a $POIAgent_{Obs(2)}$ lacks by comparison with a $POIAgent_{Obs(1)}$ is listed in Figure 115. Similarly, the deficit in knowledge for a $POIAgent_{Obs(1)}$ by comparison with a $POIAgent_{Obs(2)}$ is listed in Figure 116.

From these comparisons, it is clear that a $POIAgent_{Obs(1)}$ can represent only those blocks worlds variants in which each hand is holding a block and each table is supporting many blocks. A $POIAgent_{Obs(1)}$ cannot represent stacks of blocks, because it has not observed the `on` relationship. For this reason, it is unable to induce the `stack` and `unstack` operators. At first sight, one might think that a $POIAgent_{Obs(1)}$ can still induce the `pickUp` and `putDown` operators. However, to put a block down, a hand must become empty. Since a $POIAgent_{Obs(1)}$ does not know about the `handempty` relationship, putting a block down cannot be achieved. Therefore, a $POIAgent_{Obs(1)}$ cannot induce any of the Nilsson (1980) operators.

About object-classes:
(none)

About relationships:
```
[holding ?Hand ?Block]
```

About constraints:
```
IF holding ?Hand1 ?Block1 AND holding ?Hand2 ?Block1 THEN INVALID.
IF holding ?Hand1 ?Block1 AND holding ?Hand1 ?Block2 THEN INVALID.
IF holding ?Hand1 ?Block1 AND ontable ?Block1 ?Table1 THEN INVALID.
IF holding ?Hand1 ?Block1 AND clear ?Block1 THEN INVALID.
```

About unuseability of constraints:
```
IF ontable ?Block1 ?Table1 AND ontable ?Block2 ?Table1 THEN INVALID.
IF ontable ?Block1 ?Table1 AND clear ?Block1 THEN INVALID.
```

Figure 115: Knowledge that $POIAgent_{Obs(2)}$ Lacks, Compared to $POIAgent_{Obs(1)}$.

About object-classes:
(none)

About relationships:
```
[handempty ?Hand]
[on ?Block ?Block]
```

About constraints:
```
IF on ?Block1 ?Block1 THEN INVALID.
IF on ?Block1 ?Block2 AND on ?Block1 ?Block3 THEN INVALID.
IF on ?Block1 ?Block2 AND on ?Block3 ?Block2 THEN INVALID.
IF on ?Block1 ?Block2 AND on ?Block2 ?Block1 THEN INVALID.
IF on ?Block1 ?Block2 AND ontable ?Block1 ?Table1 THEN INVALID.
IF on ?Block1 ?Block2 AND clear ?Block2 THEN INVALID.
```

Figure 116: Knowledge that $POIAgent_{Obs(1)}$ Lacks, Compared to $POIAgent_{Obs(2)}$.

A $POIAgent_{Obs(2)}$ cannot induce any of Nilsson's (1980) planning operators, because it has not observed the `holding` relationship. Furthermore, consideration shows that a $POIAgent_{Obs(2)}$ can represent only those blocks worlds variants in which hands are always empty and blocks are found only in stacks. Any world in which any stack contained just one block would be impossible, otherwise the constraint:

```
IF ontable ?Block1 ?Table1 AND clear ?Block1 THEN INVALID
```

would be violated. There could only be one stack per table, otherwise the constraint:

```
IF ontable ?Block1 ?Table1 AND ontable ?Block2 ?Table1 THEN
INVALID
```

would be violated. In particular, the one-block world would be impossible because the block could be neither in the hand nor in a one-block stack on the table.

5.3.2 Effects of Presentation Order

It can be seen that the order of presentation of the observations determines what intermediate information is generated by a POIAgent. In particular, a $POIAgent_{Obs(2)}$ "knows" that the following constraints are USEABLE:

```
IF ontable ?Block1 ?Table1 AND ontable ?Block2 ?Table1 THEN
INVALID.

IF ontable ?Block1 ?Table1 AND clear ?Block1 THEN INVALID.
```

By contrast, a $POIAgent_{Obs(1)}$ "knows" that these constraints are UNUSEABLE.

5.3.3 What Both Observations Add

In the same way, it is possible to predict what the presentation of both observations to one POIAgent adds to presenting the observations to two separate POIAgents. This is again best expressed as a lack of knowledge, but this time resulting from distributing the observations over the two POIAgents. Figure 117 lists the simple union of the knowledge of a $POIAgent_{Obs(1)}$ with that of a $POIAgent_{Obs(2)}$.

About USEABLE constraints:

```
IF holding ?Hand1 ?Block1 AND holding ?Hand2 ?Block1 THEN INVALID.
IF holding ?Hand1 ?Block1 AND holding ?Hand1 ?Block2 THEN INVALID.
IF holding ?Hand1 ?Block1 AND ontable ?Block1 ?Table1 THEN INVALID.
IF holding ?Hand1 ?Block1 AND clear ?Block1 THEN INVALID.
IF ontable ?Block1 ?Table1 AND ontable ?Block1 ?Table2 THEN INVALID.
IF on ?Block1 ?Block1 THEN INVALID.
IF on ?Block1 ?Block2 AND on ?Block1 ?Block3 THEN INVALID.
IF on ?Block1 ?Block2 AND on ?Block3 ?Block2 THEN INVALID.
IF on ?Block1 ?Block2 AND on ?Block2 ?Block1 THEN INVALID.
IF on ?Block1 ?Block2 AND ontable ?Block1 ?Table1 THEN INVALID.
IF on ?Block1 ?Block2 AND clear ?Block2 THEN INVALID.
```

About UNUSEABLE constraints:

```
IF on ?Block1 ?Block2 AND clear ?Block1 THEN INVALID.
IF on ?Block1 ?Block2 AND on ?Block2 ?Block3 THEN INVALID.
IF on ?Block2 ?Block1 AND ontable ?Block1 ?Table1 THEN INVALID.
```

About constraints marked both USEABLE and UNUSEABLE:

```
IF ontable ?Block1 ?Table1 AND ontable ?Block2 ?Table1 THEN INVALID.
IF ontable ?Block1 ?Table1 AND clear ?Block1 THEN INVALID.
```

Figure 117: Union of Knowledge of $POIAgent_{Obs(1)}$ and of $POIAgent_{Obs(2)}$.

Comparison of Figure 117 with Figure 114 shows that the knowledge lacking as a result of its distribution concerns the constraints:

```
IF holding ?Hand1 ?Block1 AND handempty ?Hand1 THEN INVALID.
IF holding ?Hand1 ?Block1 AND on ?Block1 ?Block2 THEN INVALID.
IF holding ?Hand1 ?Block1 AND on ?Block2 ?Block1 THEN INVALID.
```

Moreover, the "knowledge" of the useability of the following constraints is conflicting:

```
IF ontable ?Block1 ?Table1 AND ontable ?Block2 ?Table1 THEN
INVALID.
```

```
IF ontable ?Block1 ?Table1 AND clear ?Block1 THEN INVALID.
```

A recipient agent which simply formed the union of knowledge would believe that, in the blocks world, it was permissible to:

- hold several blocks in a hand at the same time.
- pick up stacks of blocks, either by taking hold of the top-most block or by lifting a block lower down in the stack.

Depending on whether or not the agent detected the conflicts and, if it did, how it resolved them, the agent might believe that:

- a table can only support a single stack of blocks, and/or
- stacks on the table can only consist of a single block.

Clearly, it is not enough merely for the recipient to form the union of the sender's list of constraints with its own list. The recipient agent must perform some cognitive work to combine the information it has received with its own information. I shall term this cognitive work *assimilation*[57]. Assimilation includes resolving ambiguities in the useability of constraints. Since the POI algorithm's default rule is that a constraint is USEABLE, information showing that the constraint is in fact UNUSEABLE overrides the default. In other words, ambiguity is resolved by preserving falsity.

As well as resolving ambiguities, the recipient must identify constraints which neither the sender nor the recipient will have identified individually. These additional constraints arise from the pairing of a relationship-class identified by one agent only with another relationship-class identified only by the other agent. This implies that transmitting constraints is not sufficient. It is also necessary for the sender agent to transmit its list of relationship-classes (and, by extension, its list of object-classes). In sum, assimilation involves:

- Combining the sender's lists of object-classes, relationship-classes and constraints with the recipient's own lists.
- Identifying ambiguities in the useability of constraints and resolving them by preserving falsity.
- Identifying additional constraints arising from the pairing of a relationship-class identified by one agent only with relationship-classes identified only by the other agent. This aspect of assimilation is a subset of the functionality needed for POI Part 1.

---

[57] Lefkowitz and Lesser (1990) refer to assimilation as the knowledge acquisition process in which an existing knowledge base is modified to incorporate new information obtained from a domain expert. I generalise their use of the term to applications other than knowledge acquisition and to sender agents other than domain experts.

5.4 EXPERIMENT RESULTS

In this section, the results of the single-agent and multi-agent learning experiments are compared with their respective predictions.

5.4.1 Results of Single-Agent Learning Experiments

Two Message-Based Architecture runs were made in which there was one POIAgent. The procedure followed in each run was as laid down in section 5.1 for single-agent learning. A session trace was captured in each run. In one run ("OBS12"), the POIAgent was presented with Observation (1) first, followed by Observation (2). In the other run ("OBS21"), the order in which the observations were presented was reversed.

```
Rules in knowledge base of POIAgent1.block are:

Text of POIAgent1.block-rule1 (NOT useable) is:
 IF ontable ?POIAgent1.block1 ?POIAgent1.table1
 AND clear ?POIAgent1.block1
 THEN invalid

Text of POIAgent1.block-rule2 is:
 IF ontable ?POIAgent1.block1 ?POIAgent1.table1
 AND ontable ?POIAgent1.block1 ?POIAgent1.table2
 THEN invalid

Text of POIAgent1.block-rule3 is:
 IF holding ?POIAgent1.hand1 ?POIAgent1.block1
 AND clear ?POIAgent1.block1
 THEN invalid

Text of POIAgent1.block-rule4 is:
 IF holding ?POIAgent1.hand1 ?POIAgent1.block1
 AND ontable ?POIAgent1.block1 ?POIAgent1.table1
 THEN invalid

Text of POIAgent1.block-rule5 is:
 IF holding ?POIAgent1.hand1 ?POIAgent1.block1
 AND holding ?POIAgent1.hand2 ?POIAgent1.block1
 THEN invalid
```

Figure 118: Actual Constraints for $POIAgent_{Obs(1)}$'s Block.

The session traces were inspected. The lists of objects, relationships and constraints noted during the session were compared with the predicted lists of objects, relationships and constraints. Figure 118 is an extract from the OBS12 session trace showing the constraints identified by the $POIAgent_{Obs(1)}$ for the Block object-class. Figure 119 shows the constraints it identified for the Hand and Table object-classes. Figure 120, Figure 121 and Figure 122 are the equivalent extracts from the OBS21 session trace for the $POIAgent_{Obs(2)}$, and Figure 123, Figure 124 and Figure 125 are the extracts from the OBS12 session trace for the $POIAgent_{Obs(1\&2)}$. The OBS21 extracts for the $POIAgent_{Obs(1\&2)}$ were the same.

In every case, the actual lists matched the corresponding predicted lists. Minor differences were noted, e.g., the constraints were generated in a different order to that predicted, the antecedent clauses in some constraints were in the reverse order to that predicted, or variables were (consistently) given different names in some of the constraints. None of these differences would have affected Part 2 of

the POI algorithm.

```
Rules in knowledge base of POIAgent1.hand are:

Text of POIAgent1.hand-rule1 is:
 IF holding ?POIAgent1.hand1 ?POIAgent1.block1
 AND holding ?POIAgent1.hand1 ?POIAgent1.block2
 THEN invalid

Rules in knowledge base of POIAgent1.table are:

Text of POIAgent1.table-rule1 (NOT useable) is:
 IF ontable ?POIAgent1.block1 ?POIAgent1.table1
 AND ontable ?POIAgent1.block2 ?POIAgent1.table1
 THEN invalid
```

Figure 119: Actual Constraints for $POIAgent_{Obs(1)}$'s Hand and Table.

```
Rules in knowledge base of POIAgent1.block are:

Text of POIAgent1.block-rule1 is:
 IF on ?POIAgent1.block1 ?POIAgent1.block1
 THEN invalid

Text of POIAgent1.block-rule2 (NOT useable) is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND clear ?POIAgent1.block1
 THEN invalid

Text of POIAgent1.block-rule3 is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND clear ?POIAgent1.block2
 THEN invalid

Text of POIAgent1.block-rule4 is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND on ?POIAgent1.block1 ?POIAgent1.block4
 THEN invalid

Text of POIAgent1.block-rule5 (NOT useable) is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND on ?POIAgent1.block3 ?POIAgent1.block1
 THEN invalid

Text of POIAgent1.block-rule6 (NOT useable) is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND on ?POIAgent1.block2 ?POIAgent1.block4
 THEN invalid

Text of POIAgent1.block-rule7 is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND on ?POIAgent1.block3 ?POIAgent1.block2
 THEN invalid
```

Figure 120: Actual Constraints 1 to 7 for $POIAgent_{Obs(2)}$'s Block.

```
Text of POIAgent1.block-rule8 (NOT useable) is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND on ?POIAgent1.block1 ?POIAgent1.block2
 THEN invalid

Text of POIAgent1.block-rule9 is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND on ?POIAgent1.block2 ?POIAgent1.block1
 THEN invalid

Text of POIAgent1.block-rule10 is:
 IF ontable ?POIAgent1.block1 ?POIAgent1.table1
 AND clear ?POIAgent1.block1
 THEN invalid

Text of POIAgent1.block-rule11 is:
 IF ontable ?POIAgent1.block1 ?POIAgent1.table1
 AND on ?POIAgent1.block1 ?POIAgent1.block3
 THEN invalid

Text of POIAgent1.block-rule12 (NOT useable) is:
 IF ontable ?POIAgent1.block1 ?POIAgent1.table1
 AND on ?POIAgent1.block2 ?POIAgent1.block1
 THEN invalid

Text of POIAgent1.block-rule13 is:
 IF ontable ?POIAgent1.block1 ?POIAgent1.table1
 AND ontable ?POIAgent1.block1 ?POIAgent1.table2
 THEN invalid
```

Figure 121: Actual Constraints 8 to 13 for $\text{POIAgent}_{\text{Obs(2)}}$'s Block.

```
Rules in knowledge base of POIAgent1.hand are:
   - (None)

Rules in knowledge base of POIAgent1.table are:

Text of POIAgent1.table-rule1 is:
 IF ontable ?POIAgent1.block1 ?POIAgent1.table1
 AND ontable ?POIAgent1.block2 ?POIAgent1.table1
 THEN invalid
```

Figure 122: Actual Constraints for $\text{POIAgent}_{\text{Obs(2)}}$'s Hand and Table.

```
Rules in knowledge base of POIAgent1.block are:

Text of POIAgent1.block-rule1 (NOT useable) is:
 IF ontable ?POIAgent1.block1 ?POIAgent1.table1
 AND clear ?POIAgent1.block1
 THEN invalid

Text of POIAgent1.block-rule2 is:
 IF ontable ?POIAgent1.block1 ?POIAgent1.table1
 AND ontable ?POIAgent1.block1 ?POIAgent1.table2
 THEN invalid

Text of POIAgent1.block-rule3 is:
 IF holding ?POIAgent1.hand1 ?POIAgent1.block1
 AND clear ?POIAgent1.block1
 THEN invalid

Text of POIAgent1.block-rule4 is:
 IF holding ?POIAgent1.hand1 ?POIAgent1.block1
 AND ontable ?POIAgent1.block1 ?POIAgent1.table1
 THEN invalid

Text of POIAgent1.block-rule5 is:
 IF holding ?POIAgent1.hand1 ?POIAgent1.block1
 AND holding ?POIAgent1.hand2 ?POIAgent1.block1
 THEN invalid

Text of POIAgent1.block-rule6 is:
 IF on ?POIAgent1.block1 ?POIAgent1.block1
 THEN invalid

Text of POIAgent1.block-rule7 (NOT useable) is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND clear ?POIAgent1.block1
 THEN invalid

Text of POIAgent1.block-rule8 is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND clear ?POIAgent1.block2
 THEN invalid

Text of POIAgent1.block-rule9 is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND ontable ?POIAgent1.block1 ?POIAgent1.table1
 THEN invalid

Text of POIAgent1.block-rule10 (NOT useable) is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND ontable ?POIAgent1.block2 ?POIAgent1.table1
 THEN invalid

Text of POIAgent1.block-rule11 is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND holding ?POIAgent1.hand1 ?POIAgent1.block1
 THEN invalid
```

Figure 123: Actual Constraints 1 to 11 for $POIAgent_{Obs(1\&2)}$'s Block.

```
Text of POIAgent1.block-rule12 is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND holding ?POIAgent1.hand1 ?POIAgent1.block2
 THEN invalid

Text of POIAgent1.block-rule13 is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND on ?POIAgent1.block1 ?POIAgent1.block4
 THEN invalid

Text of POIAgent1.block-rule14 (NOT useable) is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND on ?POIAgent1.block3 ?POIAgent1.block1
 THEN invalid

Text of POIAgent1.block-rule15 (NOT useable) is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND on ?POIAgent1.block2 ?POIAgent1.block4
 THEN invalid

Text of POIAgent1.block-rule16 is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND on ?POIAgent1.block3 ?POIAgent1.block2
 THEN invalid

Text of POIAgent1.block-rule17 (NOT useable) is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND on ?POIAgent1.block1 ?POIAgent1.block2
 THEN invalid

Text of POIAgent1.block-rule18 is:
 IF on ?POIAgent1.block1 ?POIAgent1.block2
 AND on ?POIAgent1.block2 ?POIAgent1.block1
 THEN invalid
```

Figure 124: Actual Constraints 12 to 18 for $POIAgent_{Obs(1\&2)}$'s Block.

```
Rules in knowledge base of POIAgent1.hand are:

Text of POIAgent1.hand-rule1 is:
 IF holding ?POIAgent1.hand1 ?POIAgent1.block1
 AND holding ?POIAgent1.hand1 ?POIAgent1.block2
 THEN invalid

Text of POIAgent1.hand-rule2 is:
 IF handempty ?POIAgent1.hand1
 AND holding ?POIAgent1.hand1 ?POIAgent1.block1
 THEN invalid

Rules in knowledge base of POIAgent1.table are:

Text of POIAgent1.table-rule1 (NOT useable) is:
 IF ontable ?POIAgent1.block1 ?POIAgent1.table1
 AND ontable ?POIAgent1.block2 ?POIAgent1.table1
 THEN invalid
```

Figure 125: Actual Constraints for $POIAgent_{Obs(1\&2)}$'s Hand and Table.

5.4.2 Results of Multi-Agent Learning Experiments

Two pairs of Message-Based Architecture runs were made in which there were two POIAgents. The procedure followed in each run was as laid down in Section 5.2 for multi-agent learning. A session trace was captured in each run. One POIAgent was presented with Observation (1), and the other POIAgent was presented with Observation (2). In one pair of runs, the $\text{POIAgent}_{\text{Obs(1)}}$ sent its lists of objects, relationships and constraints to the $\text{POIAgent}_{\text{Obs(2)}}$. In the other pair of runs, the $\text{POIAgent}_{\text{Obs(2)}}$ sent its lists of objects, relationships and constraints to the $\text{POIAgent}_{\text{Obs(1)}}$. Comparison confirmed that the $\text{POIAgents}_{\text{Obs(1)}}$ and $\text{POIAgents}_{\text{Obs(2)}}$ in the multi-agent learning runs identified the same lists of objects, relationships, and constraints as the equivalent agents in the single-agent learning runs.

```
Status of POIAgent1.hand:
   - Relationships are:
      [holding ?POIAgent1.hand ?POIAgent1.block]
      [handempty ?POIAgent1.hand]
   - Actor classes are:
      POIAgent1.hand
   - Agent classes are:
      POIAgent1.block
   - Instances are:
      pOIAgent1.hand1
   - Facts database is:
      ontable ?Block1 ?Table1
      ontable ?Block2 ?Table1
   - Debugging output disabled.
(End of status report for POIAgent1.hand.)

Rules in knowledge base of POIAgent1.hand are:

Text of POIAgent1.hand-rule1 is:
 IF
   holding ?POIAgent1.hand1 ?POIAgent1.block1
 AND
   holding ?POIAgent1.hand1 ?POIAgent1.block2
 THEN
   invalid

Text of POIAgent1.hand-rule2 is:
 IF
   holding ?POIAgent1.hand1 ?POIAgent1.block1
 AND
   handempty ?POIAgent1.hand1
 THEN
   invalid
```

Figure 126: Hand's Exclusion-Rules Obtained with Assimilation.

For one run in each pair the recipient agent was provided with assimilation capabilities. With assimilation capabilities, the recipient agent in each case became a $\text{POIAgent}_{\text{Obs(1\&2)}}$. Figure 126 shows the exclusion-rules for the recipient's model of the `Hand` class. Comparison confirmed that the $\text{POIAgents}_{\text{Obs(1\&2)}}$ in the multi-agent learning runs identified the same lists of objects, relationships, and constraints as the equivalent agents in the single-agent learning runs.

For the other run in each pair the recipient agent's assimilation capabilities were inhibited. Without assimilation, the recipient merely formed the union of the $\text{POIAgent}_{\text{Obs(1)}}$'s and $\text{POIAgent}_{\text{Obs(2)}}$'s lists of constraints. The union of constraint-lists matches the predicted union shown in Figure 117. Figure 127 shows that, although the recipient's model of the `Hand` class includes the `holding` and

handempty relationship-classes, the exclusion-rule joining them has not been created.

```
Status of POIAgent1.hand:
   - Relationships are:
      [holding ?POIAgent1.hand ?POIAgent1.block]
      [handempty ?POIAgent1.hand]
   - Actor classes are:
      POIAgent1.hand
   - Agent classes are:
      POIAgent1.block
   - Instances are:
      pOIAgent1.hand1
   - Facts database is:
      ontable ?Block1 ?Table1
      ontable ?Block2 ?Table1
   - Debugging output disabled.
(End of status report for POIAgent1.hand.)

Rules in knowledge base of POIAgent1.hand are:

Text of POIAgent1.hand-rule1 is:
 IF
   holding ?POIAgent1.hand1 ?POIAgent1.block1
 AND
   holding ?POIAgent1.hand1 ?POIAgent1.block2
 THEN
   invalid
```

Figure 127: Hand's Exclusion-Rules Obtained without Assimilation.

5.5 CLOSING THE LOOP

In this section I demonstrate that the domain knowledge acquired by a $POIAgent_{Obs(1\&2)}$ supports the induction of a set of planning operators, the generation of a plan from the induced operator-set, and the successful execution of the plan to achieve the goal state. The demonstration is designed to show the blocks world passing through a novel state, i.e., a state which is equivalent neither to Observation (1) nor to Observation (2).

Each $POIAgent_{Obs(1\&2)}$ could be tested by instructing it to generate a plan containing a novel state for many different planning problems. I chose to instruct the $POIAgents_{Obs(1\&2)}$ to generate a plan from the Observation (1) state to one of the other two states equivalent to Observation (1). The goal state is given in Figure 128.

```
Goal state:
            [clear block1]
            [clear block3]
            [holding hand1 block2]
            [ontable block1 table1]
            [ontable block3 table1]
```

Figure 128: Goal which Triggers Induction.

This forces the agent to generate a plan passing through the state in which all three blocks are on the table. This state - given in Figure 129 - is novel to the $POIAgents_{Obs(1\&2)}$.

```
Novel state:
            [clear block1]
            [clear block2]
            [clear block3]
            [ontable block1 table1]
            [ontable block2 table1]
            [ontable block3 table1]
            [handempty hand1].
```

Figure 129: Novel State which Plan Passes Through.

When the $POIAgents_{Obs(1\&2)}$ were instructed to achieve the goal state, they:

- Detected the need for induction (see Figure 130).

- Performed an induction, giving the operator-set shown in Figure 131 and Figure 132. The `stack` and `unstack` operators were induced as `operator1` and `operator2`, and the `pickUp` and `putDown` operators were induced as `operator5` and `operator6`. An additional pair of `stack` and `unstack` operators were induced as `operator3` and `operator4` (see Figure 132) by generalising a different transition to the one used as the basis for `operator1` and `operator2`. Ideally, the Message-Based Architecture code should be modified so that duplicated operators are eliminated. Nevertheless, the important fact is that the full operator-set was induced.

```
pOIAgent1 receives following messages:
   from hand1: >NOT handempty hand1<
   from hand1: >NOT holding hand1 block2<
   from block1: >NOT holding hand1 block1<
   from block1: >ontable block1 table1<
   from block1: >clear block1<
   from block2: >holding hand1 block2<
   from block2: >NOT ontable block2 table1<
   from block2: >NOT clear block2<
   from table1: >NOT ontable block2 table1<
   from table1: >ontable block1 table1<
   from table1: >ontable block3 table1<
   from block3: >ontable block3 table1<
   from block3: >clear block3<

pOIAgent1 finds reports of inconsistencies;
pOIAgent1 needs to plan.

pOIAgent1 has planning capabilities.
but unable to plan because NO planning operators;
pOIAgent1 needs to induce planning operators.

pOIAgent1 has POI capabilities.

pOIAgent1 inducing planning operators.
This will take a long time (several minutes).
```

Figure 130: $POIAgent_{Obs(1\&2)}$ Detects Need for Induction.

- Generated a plan (see Figure 133).

- Executed it successfully (see Figure 134).

Regardless of the manner in which their knowledge was acquired, all $POIAgents_{Obs(1\&2)}$ successfully induced the same set of planning operators.

```
Design of operator1 ?Hand1 ?Block1 ?Block2:
   - my (filter) pre-conditions are:
   [isHand ?Hand1]
   [isBlock ?Block1]
   [isBlock ?Block2]
   - my delete-list is:
   [holding ?Hand1 ?Block1]
   [clear ?Block2]
   - my add-list is:
   [handempty ?Hand1]
   [clear ?Block1]
   [on ?Block1 ?Block2]
   - (no display text)
(End of design report for operator1 ?Hand1 ?Block1 ?Block2.)

Design of operator2 ?Hand1 ?Block1 ?Block2:
   - my (filter) pre-conditions are:
   [isHand ?Hand1]
   [isBlock ?Block1]
   [isBlock ?Block2]
   - my delete-list is:
   [handempty ?Hand1]
   [clear ?Block1]
   [on ?Block1 ?Block2]
   - my add-list is:
   [holding ?Hand1 ?Block1]
   [clear ?Block2]
   - (no display text)
(End of design report for operator2 ?Hand1 ?Block1 ?Block2.)

Design of operator5 ?Hand1 ?Block1 ?Table1:
   - my (filter) pre-conditions are:
   [isHand ?Hand1]
   [isBlock ?Block1]
   [isTable ?Table1]
   - my delete-list is:
   [holding ?Hand1 ?Block1]
   - my add-list is:
   [handempty ?Hand1]
   [clear ?Block1]
   [ontable ?Block1 ?Table1]
   - (no display text)
(End of design report for operator5 ?Hand1 ?Block1 ?Table1.)

Design of operator6 ?Hand1 ?Block1 ?Table1:
   - my (filter) pre-conditions are:
   [isHand ?Hand1]
   [isBlock ?Block1]
   [isTable ?Table1]
   - my delete-list is:
   [handempty ?Hand1]
   [clear ?Block1]
   [ontable ?Block1 ?Table1]
   - my add-list is:
   [holding ?Hand1 ?Block1]
   - (no display text)
(End of design report for operator6 ?Hand1 ?Block1 ?Table1.)
```

Figure 131: Operator-Set Induced by $POIAgent_{Obs(1\&2)}$.

```
Design of operator3 ?Hand1 ?Block1 ?Block3:
   - my (filter) pre-conditions are:
   [isHand ?Hand1]
   [isBlock ?Block1]
   [isBlock ?Block3]
   - my delete-list is:
   [holding ?Hand1 ?Block1]
   [clear ?Block3]
   - my add-list is:
   [handempty ?Hand1]
   [clear ?Block1]
   [on ?Block1 ?Block3]
   - (no display text)
(End of design report for operator3 ?Hand1 ?Block1 ?Block3.)

Design of operator4 ?Hand1 ?Block1 ?Block3:
   - my (filter) pre-conditions are:
   [isHand ?Hand1]
   [isBlock ?Block1]
   [isBlock ?Block3]
   - my delete-list is:
   [handempty ?Hand1]
   [clear ?Block1]
   [on ?Block1 ?Block3]
   - my add-list is:
   [holding ?Hand1 ?Block1]
   [clear ?Block3]
   - (no display text)
(End of design report for operator4 ?Hand1 ?Block1 ?Block3.)
```

Figure 132: Additional Operators Induced by $\text{POIAgent}_{\text{Obs(1\&2)}}$

```
INITIAL STATE is:
  [holding hand1 block1]
  [ontable block2 table1]
  [clear block2]
  [ontable block3 table1]
  [clear block3]
  [isHand hand1]
  [isBlock block1]
  [isBlock block2]
  [isTable table1]
  [isBlock block3]
GOAL STATE is:
  [holding hand1 block2]
  [ontable block1 table1]
  [clear block1]
  [ontable block3 table1]
  [clear block3]
  [isHand hand1]
  [isBlock block1]
  [isBlock block2]
  [isTable table1]
  [isBlock block3]

Planning ...
PLAN FOUND is:
  [operator5 hand1 block1 table1]
  [operator6 hand1 block2 table1]
Plan found in 196 secs.
```

Figure 133: $\text{POIAgent}_{\text{Obs(1\&2)}}$ Generates Plan using Induced Operators.

```
pOIAgent1 initiates what-if NEGOTIATION of:
  >ontable block1 table1<
  >clear block1<
  >ontable block3 table1<
  >clear block3<
  >handempty hand1<
  >clear block2<
  >ontable block2 table1<

. . .

pOIAgent1 finds agreement between objects.

pOIAgent1 (re-)setting goals from stack.

pOIAgent1 initiates what-if NEGOTIATION of:
  >holding hand1 block2<
  >ontable block1 table1<
  >clear block1<
  >ontable block3 table1<
  >clear block3<

. . .

definer1 ACHIEVES GOALS:
   holding hand1 block2
   ontable block1 table1
   clear block1
   ontable block3 table1
   clear block3

definer1's stack is EMPTY.
```

Figure 134: $POIAgent_{Obs(1\&2)}$ Successfully Executes Generated Plan.

5.6 CHAPTER 5 CONCLUSIONS

This chapter has described a series of experiments with the multi-agent implementation of the POI algorithm: the Message-Based Architecture testbed. The experiment series was designed for closed-loop testing of the algorithm. The loop was closed from the algorithm's outputs back to its inputs by embedding the POI algorithm, together with reactive and deliberative planning functionalities, in a *POIAgent*.

POIAgents were placed in an environment consisting of several other simpler agents, which simulated the POIAgent's problem domain. At the start of each run, the POIAgents were *naive*, i.e., they had no prior domain knowledge whatsoever. Each POIAgent was given a series of goals to achieve by interacting with the simpler agents in its environment. The goal-series were designed so that the POIAgents acquired knowledge about their environment, induced a set of planning operators from the acquired knowledge, generated a plan using the induced operators, and executed that plan successfully. In short, the POIAgents *learned-by-doing* (Anzai and Simon, 1979).

Some experiments were done with a single POIAgent. Other experiments were done with two POIAgents. The former experiments investigated *single-agent learning in a multi-agent environment*, and the latter investigated *multi-agent learning*. In the multi-agent learning case, the POIAgents cooperated by exchanging information they had acquired about their own problem domains. In short, the POIAgents also *learned-by-being-told* by other POIAgents.

Hypothesis validation was performed on the experiment results by comparing the series of lists of object-classes, relationship-classes and interrelationship constraints identified by the POIAgents as they

achieved the series of goals. In the absence of an interactive planning operator oracle, the lists were predicted manually. Predicted lists were obtained for all orderings of the series of goals. I showed that there were intermediate predictions that were dependent on the order of presentation of the goals, but that the predicted lists at the end of the series of goals were independent of the presentation ordering. This order-dependent and order-independent behaviour was found, as predicted, in the experiment runs.

For experimentation purposes, the blocks world was used as the simulated problem-solving domain. Each block, hand and table was represented as a simple *non-intentional* agent (Grant, 1992b). A pair of 3-blocks-world states have been found which, when described to a single POIAgent, caused it to induce correctly the complete, Nilsson (1980) set of blocks-world planning operators. To avoid any suggestion that the induction process was making use of the pair of world states as a sequence (i.e., as a plan-segment), the descriptions were presented to the POIAgent as two, separate 3-blocks worlds. The POIAgent was merely given the goal of initialising the two blocks worlds; this was achievable purely by reactive means. In the process, the POIAgent learned enough about the blocks world to induce a set of planning operators. Induction was then triggered by instructing the POIAgent to solve a further goal which requires the generation of a plan. Lacking planning operators, the POIAgent first had to induce a set of planning operators before it could generate the plan.

In the multi-agent learning experiments, each of the two POIAgents was presented with one of the world-state descriptions. Effectively, each POIAgent observed a separate 3-blocks world. I found that the sum of the knowledge extracted by the two POIAgents individually was less than the knowledge extracted by a single POIAgent from the same pair of observations. In essence, knowledge had been lost by distributing it across multiple agents. To counter this loss of knowledge, I enhanced the POIAgents' capabilities to be able to exchange knowledge and, more importantly, to *assimilate* the knowledge gained from another POIAgent with its own knowledge.

Two issues arose:

- *In what form should the knowledge be exchanged?* Since the POIAgents do not retain the raw observations (cf. cases) and might not know enough to induce a set of planning operators, I chose to give them the capability to exchange lists of constraints. I found the constraints had to be supplemented by an entity-relationship domain model, i.e., by the exchange of lists of object-classes and relationship-classes.

- *What must the recipient POIAgent do to assimilate knowledge obtained from another POIAgent?* Merely adding the lists of entities, relationships and constraints to its own lists was not enough. In the experiment runs, some of the information was even contradictory. I found that the recipient POIAgent had to execute part of the POI algorithm to assimilate knowledge gained from other agents. There was still a net gain in cognitive effort for the recipient POIAgent, compared to having to observe a second world. Moreover, the agents providing the lists of entities, relationships and constraints did not need the induction capabilities.

Extracts from the traces of the experiment runs are reproduced in this chapter. The extracts show that:

- the lists of constraints which were generated during single- and multi-agent learning differed from the predicted lists only in unimportant respects, e.g., the ordering of constraints within a list, the ordering of antecedent clauses within a constraint, or the naming of variables within a constraint.

- the list of constraints generated when part of the series of goals had been achieved was

dependent on the order in which the goals were presented, as predicted.

- the list of constraints generated when the complete series of goals had been achieved was independent of the order in which the goals were presented, as predicted.

- without assimilation, the recipient POIAgent in the multi-agent learning experiments merely forms the union of the lists of constraints.

- with assimilation, the recipient POIAgent in the multi-agent learning experiments identifies the full set of constraints, as in single-agent learning.

- POIAgents with a full set of object-classes, relationship-classes and interrelationship constraints, whether obtained by single-agent learning or multi-agent learning with assimilation, were able to:

    - detect the need for induction.

    - induce a set of planning operators which differed only in unimportant respects from the operator-set in (Nilsson, 1980).

    - generate a plan using the induced operator-set.

    - execute the plan successfully.

I have made several research contributions in this chapter. I have demonstrated that:

- the POI algorithm has been successfully tested in a multi-agent environment. In particular, these tests have proven Part (1) ("Acquisition").

- the POI algorithm has been successfully integrated with plan generation and execution to close the loop from the POI outputs to its inputs.

- the POI algorithm has an application in multi-agent systems to give agents the capability of learning-by-doing.

- Lefkowitz and Lesser's (1990) concept of knowledge assimilation can be generalised to contexts other than knowledge acquisition and to senders other than domain experts.

- knowledge assimilation can be implemented by re-using part of the POI algorithm.

- learning-by-doing can be integrated with learning-by-being-told by means of knowledge assimilation.

6 CONCLUSIONS

6.1 RESEARCH RESULTS

6.1.1 Research Objectives

This thesis has addressed the research question: "Where do planning operators come from?". At present, the developers of a planning system are responsible for providing a suitable set of planning operators for each problem domain. However, this begs a series of further questions concerning the completeness, correctness and precision of planning operators developed by (manual) knowledge acquisition methods. Most importantly, the Knowledge Acquisition Bottleneck applies.

The objective of my research has been to circumvent the Knowledge Acquisition Bottleneck by automated learning of knowledge-based planning operators. I have shown that STRIPS-style planning operators can be induced *ab initio* from unordered lists of domain objects, inter-object relationships, and interrelationship constraints. Moreover, such lists can be compiled from unordered observations of non-adjacent states in the domain, giving an advantage over other operator-learning algorithms.

6.1.2 Planning Operator Induction Algorithm

In this thesis, I introduce and document the Planning Operator Induction (POI) algorithm. The POI algorithm has two parts; Acquisition and Induction. The core of the algorithm is the second part, which performs the induction of planning operators from lists of domain objects, relationships and constraints using a variant of Mitchell's (1982) version space and candidate elimination algorithm. The Acquisition part can be added where it is necessary to compile such lists from unordered observations of domain states.

6.1.3 Implementations

There are two implementations of the POI algorithm. Both are implemented in the Smalltalk/V object-oriented programming language and run on PCs under MS-DOS. One implementation is a single-agent system, known as the Dutch Utilisation Centre's Activity Scheduling System (DUC-ASS). DUC-ASS implements Part 2 (Induction), plus some enhancements designed to counter the combinatorial explosion inherent in the version space and candidate elimination algorithm. The other implementation is a multi-agent system, known as the Message-Based Architecture testbed. The Message-Based Architecture testbed implements both parts of the POI algorithm, but not as faithfully as the DUC-ASS.

The Activity Scheduling System user defines a domain in terms of the classes of domain objects and inter-object relationships. As each relationship is defined, the Activity Scheduling System generates all meaningful constraints with previously-defined relationships, interacting with the user to elicit whether or not the constraint is to hold in the domain. Prior to triggering induction, the user defines a set of instances for each object-class. For research purposes, the user can trigger each step in (Part 2 of) the POI algorithm individually. This allows intermediate results to be inspected and, if the user desires, to be marked as being unuseable for subsequent steps. A set of planning operators is output

in the final step. The program is instrumented to obtain statistics such as computation time per step and memory space used.

Additional functionality has been implemented in the Activity Scheduling System to extract plans, schedules and resource profiles, and to depict the domain model and version space graphically as an entity-relationship diagram and a state-transition network, respectively. Extension of this additional functionality is foreseen to plan execution, to execution monitoring, to diagnosis of execution failures in terms of constraint violations, and to generation of recovery plans by re-induction of the version space with relaxed constraints. This additional functionality and its foreseen extensions are all outside the scope of the research reported in this thesis.

The Message-Based Architecture testbed enables the interactions between a multi-agent problem domain and one or more additional agents with built-in POI capabilities (*POIAgents*) to be investigated. The user initialises the testbed by defining the problem domain and the POIAgents. By default, newly-defined POIAgents have no knowledge of any problem domain. The user triggers interaction between the POIAgents and the problem domain by instructing the POIAgents to achieve a series of goals to achieve in the problem domain. The goals can be designed so as to cause the POIAgents to induce a set of planning operators. In achieving the goals, the POIAgents observe a variety of states of the problem domain. These observations become the input to the POI algorithm. The planning operators output by the POI algorithm are used by the POIAgents in plan generation, and the generated plan is then executed on the problem domain.

6.1.4 Experiments

The POI implementations have been tested using eight domains. The domains vary in complexity from one to 24 object-classes. This thesis documents experiments performed for three illustrative domains: Piano-Playing, Blocks World, and High Performance Capillary Electrophoresis. The Piano-Playing domain, comprising two object-classes, is used to introduce POI concepts, to illustrate the workings of the POI algorithm, together with its underlying ontology, and to investigate the sensitivity of the POI algorithm to variation in its inputs. In particular, the piano-playing domain is used to demonstrate that even the simplest of domains may suffer an immense combinatorial explosion. The effectiveness of various countermeasures is shown.

Blocks world experiments have been performed using both POI implementations. Using DUC-ASS, experiments have been done to reproduce the Blocks World operator-sets published in seven AI textbooks and to demonstrate the value of introducing object-class inheritance. Sensitivity analysis of the Blocks World has confirmed the sensitivity analysis done for the Piano-Playing domain.

Experimentation with the 24-object-class High Performance Capillary Electrophoresis (HPCE) domain demonstrates conclusively that the POI algorithm is capable of handling complex, real-world domains. Moreover, novel planning operators have been induced, which other operator-learning algorithms would be unable to generate. A domain expert judged the POI algorithm's output as being representative for the domain.

Using the Message-Based Architecture testbed, a series of experiments has been done in which the problem domain modelled two separate blocks worlds. One experiment was done with a single POIAgent. This experiment was designed to show that it is possible for a POIAgent to acquire

sufficient information to induce a full set of operators merely by initialising the two blocks worlds[58]. Given a goal state to achieve, the POIAgent induces the operator-set and generates a plan which, on execution, passes through a state which the POIAgent has not been shown. This experiment conclusively demonstrates that the POI algorithm is capable of inducing novel domain knowledge from observed states. To ensure that sequencing information plays no part in the induction, observed domain states were chosen so that no pair of states were adjacent, i.e., shared a transition in common. Moreover, the experiment run was repeated with the two initialisations reversed, giving identical results.

A second set of Message-Based Architecture testbed experiments was done with two POIAgents. The observations were distributed between the two POIAgents so that one POIAgent initialised one blocks world and the other POIAgent initialised the other blocks world. Neither POIAgent had sufficient information on its own to induce any planning operators. The POIAgents were then given the functionality to exchange lists of observed domain objects, relationships and constraints, i.e., the outputs of Part 1 of the POI algorithm. A POIAgent which, on receiving another POIAgent's lists, performed the simple union of the lists of observed domain objects, relationships and constraints was still unable to induce any planning operators. It was found necessary to provide additional functionality for the *recipient* POIAgent to assimilate other POIAgents' lists with its own. The knowledge assimilation functionality was readily implementable as a specialisation of Part 2 of the POI algorithm. Experiments were done with assimilation enabled and disabled. With assimilation enabled, the recipient POIAgent was capable of inducing a complete and correct set of planning operators[59]. As in the first Message-Based Architecture testbed experiment, the induced operator-set was then used successfully to generate a plan which, on execution, caused the problem domain to pass through a state that was novel to the POIAgents.

6.1.5 Research Findings

The experiment results demonstrate that it is feasible to:

- automate the learning of knowledge-based planning operators. This finding confirms the results of other researchers.

- induce knowledge-based planning operators *ab initio* from unordered lists of domain objects, inter-object relationships, and interrelationship constraints. This is an advance on other operator-learning algorithms, which rely on the prior existence of plans, plan segments, or primitive operators.

- compile lists of domain objects, relationships, and constraints from unordered observations of non-adjacent domain states.

- induce planning operators for a variety of domains, including complex, real-world domains.

Furthermore, I conclude that:

---

[58] The initial states of the blocks worlds had to be carefully selected. The issue of selecting examples to optimise learning is outside the scope of my research.

[59] With correctness and completeness being judged by comparison Nilsson's (1980) operator-set, which served as an oracle.

- The POI algorithm distinguishes itself in two ways from previous methods of automating the learning of planning operators:
  - The POI algorithm does not require any sequencing information.
  - The POI algorithm uses an induction algorithm.
- The POI algorithm builds on previous research, notably Mitchell's (1982) version space and candidate elimination algorithm.
- Like the version space and candidate elimination algorithm, the POI algorithm suffers from combinatorial explosion. Within-domain complexity is consistent with exponential behaviour, but across-domain is consistent with a third-order polynomial.
- While the combinatorial explosion cannot be eliminated, its effects can be countered to such an extent that the POI algorithm can be applied to complex, real-world domains, such as the High Performance Capillary Electrophoresis instrument.
- The countermeasures developed to combat the combinatorial explosion in the POI algorithm are applicable to other algorithms which suffer from a combinatorial explosion. In particular, they are potentially applicable to the version space and candidate elimination algorithm and its variants. This finding is justified by the fact that the countermeasures are specific to the underlying POI ontology, which is independent both of the domain and of the induction algorithm. This finding must be confirmed by future research outside the scope of this thesis to apply the countermeasures in inducing other data-structures.

6.2 CONTRIBUTIONS OF MY RESEARCH

My research contributes to knowledge in the fields of knowledge-based planning, multi-agent systems, machine learning, general AI, and software engineering.

I have contributed to the knowledge-based planning field by showing that:

- the Knowledge Acquisition Bottleneck in planning can be countered by the automated learning of domain knowledge.
- planning operators can be induced.
- a coherent rationale can be found for the diversity of blocks world operator-sets to be found in AI textbooks.
- a disciplined approach can be taken to acquiring domain knowledge in the form of planning operators.

I have contributed to the multi-agent systems field by:

- showing that learning-by-doing can be integrated with learning-by-being-told.
- extending and implementing Lefkowitz and Lesser's (1990) concepts of knowledge assimilation.

I have contributed to the machine learning field by:

- finding an alternative approach to inducing disjunctive concepts by means of the version space and candidate elimination algorithm.

- analysing the complexity of the version space and candidate elimination algorithm, as applied to version spaces of binary inter-object relationship-instances, both theoretically and empirically.

- developing a set of domain-independent countermeasures to the combinatorial explosion suffered by the version space and candidate elimination algorithm.

- developing an algorithm for inducing new domain theories, expressed as planning operators, from unsequenced state descriptions/observations.

I have contributed to general AI by developing an ontology which, although tailored to the needs of inducing planning operators, could be applied more widely.

I have contributed to software engineering by developing an AI-based algorithm for obtaining the state-transition network for a domain from an extended entity-relationship model of that domain.

6.3 DIRECTIONS FOR FUTURE RESEARCH

There are many possible directions for future research:

- More experiments could be done, e.g., for additional domains. In particular, an investigation should be done into inducing planning operators for problem domains containing intentional agents, i.e., agents which are themselves capable of planning, learning, and/or planning operator induction.

- The existing POI implementations could be employed in new application areas, e.g., as a tool to support knowledge acquisition or, embedded in a Computer-Aided Software Engineering tool, to support conventional software development.

- The POI implementations could be enhanced. Enhancements could take the form of graphic user interfaces, the additional functionality foreseen for the DUC-ASS, or employing an alternative induction algorithm to the version space and candidate elimination algorithm.

- The POI algorithm could be recast as an incremental algorithm.

- The POI ontology and algorithm could be extended to cover metric attributes and/or imprecise information. This would enable POI to be extended to scheduling, design, and uncertain domains.

- The countermeasures developed to combat the combinatorial explosion in the POI algorithm could be applied to other algorithms in induction, machine learning, or other fields of AI.

- The ontology developed for POI could be extended to classes of problem-solving, e.g., classification, diagnosis, etc.

- The POI ontology should be investigated to identify the justification underlying the empirical adoption of binary relationships and binary constraints in this research.

- Research is needed into a methodology for developing knowledge-based planning systems. The need for a methodology is demonstrated by the absence in several leading AI textbooks of a clear definition of the domain for which planning operators are presented.

- Research is needed into ways of testing and validating sets of planning operators. There is a lack of accepted oracles and/or benchmarks. The use of formal methods and domain simulations are possibilities.

6.4 SIGNIFICANCE OF MY RESEARCH

My research has significance both in AI and in software engineering. For researchers in the AI area of knowledge-based planning, the research is significant in that it is now practicable to induce a set of planning operators from a domain model, without needing prior examples of domain behaviour. This opens the possibility of speeding up the generation of operator-sets, with the attached guarantee of their consistency with the domain model input. Moreover, I have faced up to the problem of combinatorial explosion, and have developed some countermeasures which may be more widely applicable in AI research.

For software engineers, I have developed an algorithm which can automatically generate state-transition networks from an enhanced entity-relationship model. This opens up the possibility of saving effort in software design, with the added guarantee that the state-transition network and entity-relationship model are consistent with one another.

References

Allen, J. F. (1990). Formal Models of Planning. In Allen, Hendler, and Tate (1990), 50-54.

Allen, J. F., Hendler, J., and Tate, A. (eds) (1990). *Readings in Planning*. Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, ISBN 1-55860-130-9.

Allen, J. F., Kautz, H. A., Pelvin, R. N., and Tenenberg, J. D. (1991). *Reasoning about Plans*. Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, ISBN 1-55860-137-6.

Alterman, R., and Zito-Wolf, R. (1993). Agents, Habitats, and Routine Behaviour. *Proceedings of the 1993 International Joint Conference on Artificial Intelligence* (IJCAI-93), Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, ISBN 1-55860-300-X, Volume 1, 305-310.

Ambros-Ingerson, J. A., and Steel, S. (1988). Integrating Planning, Execution and Monitoring. *Proceedings of the 1988 AAAI Conference*, 83-88. Also published in Allen, Hendler and Tate (1990), 735-740.

Angluin, D., and Smith, C. H. (1983). Inductive Inference: Theory and Methods. *ACM Computing Surveys*, **15**, 237-269.

Anzai, Y., and Simon, H. A. (1979). The Theory of Learning by Doing. *Psychological Review*, **86**[60], 124-140.

Beckman Instruments Inc. (1989). *P/ACE System 2000 Instrument Manual*. Beckman Instruments Nederland bv, Mijdrecht, The Netherlands.

Blum, B. I. (1992). *Software Engineering: A Holistic View*. Oxford University Press, New York, NY, USA, ISBN 0-19-507159-X.

Blythe, J., and Mitchell, T. M. (1989). On Becoming Reactive. In Segre (1989), 255-257.

Boden, M. A. (ed.) (1990). *The Philosophy of Artificial Intelligence*. Oxford University Press, Oxford, England, ISBN 0-19-824854-7.

Bond, A. H., and Gasser, L. (eds) (1988). *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, ISBN 0-934613-63-X.

Bresina, J., Drummond, M., and Kedar, S. (1993). Reactive, Integrated Systems Pose New Problems for Machine Learning. In Minton (1993), 159-195.

Breuker, J. A., and Velde, W. van de (eds) (1994). *Reuseable Problem Solving Components: The CommonKADS Library for Expertise Modelling*. IOS Press, Amsterdam, The Netherlands, ISBN 90 5199 164 9.

[60] (Anzai and Simon, 1979) is often cited, incorrectly, as being in Volume 36.

Buchanan, B. G., and Mitchell, T. M. (1978). Model-Directed Learning of Production Rules. In Waterman and Hayes-Roth (1978), 297-312.

Buchanan, B. G., and Wilkins, D. C. (1993). *Readings in Knowledge Acquisition and Learning: Automating the Construction and Improvement of Expert Systems*. Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, ISBN 1-55860-163-5.

Bundy, A., Burstall, R. M., Weir, S., and Young, R. M. (1980). *Artificial Intelligence: An Introductory Course*. Revised edition, Edinburgh University Press, Scotland, ISBN 0 85224 410 X.

Bundy, A., Silver, D., and Plummer, D. (1985). An Analytical Comparison of Some Rule Learning Programs. *Artificial Intelligence*, **27**, 137-181.

Carbonell, J. G. (1989). Introduction: Paradigms for Machine Learning. *Artificial Intelligence*, **40**, 1-9.

Carbonell, J. G., and Gil, Y. (1990). Learning by Experimentation: The Operator Refinement Method. In Y. Kodratoff and R. S. Michalski (eds), (1990). *Machine Learning: An Artificial Intelligence Approach*. Volume 3, Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, 191-213 (chapter 7), ISBN 1-55860-119-8.

Chapman, D. (1987). Planning for Conjunctive Goals. *Artificial Intelligence*, **32**, 333-377.

Charniak, E., and McDermott, D. (1985). *Introduction to Artificial Intelligence*. World Student Series Edition, Addison-Wesley, Reading, MA, USA, ISBN 0-201-11946-3.

Chen, P. P.-S. (1976). The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, **1**, 9-36.

Cohen, P. R. (1991). A Survey of the Eighth National Conference on AI: Pulling Together or Pulling Apart. *AI Magazine*, **12**, 16-41.

Cohen, P. R., and Feigenbaum, E. A. (eds) (1982). *The Handbook of Artificial Intelligence*. Volume 3, Pitman Books Ltd, London, England, ISBN 0 273 08554 9.

Cohen, P. R., and Levesque, H. J. (1987). Persistence, Intention and Commitment. In Georgeff and Lansky (1987), 297-340.

Cohen, P. R., Greenberg, M. L., Hart, D. M., and Howe, A. E. (1989). *Trial by Fire: Understanding the Design Requirements for Agents in Complex Environments*. COINS Technical Report 89-61, Experimental Knowledge Systems Laboratory, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, USA.

Corkill, D. D. (1982). *A Framework for Organization Self-Design in Distributed Problem-Solving Networks*. PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, USA.

Dean, T. L., and Wellman, M. P. (1991). *Planning and Control*. Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, ISBN 1-55860-209-7.

Dean, T. L., Firby, J., and Miller, D. (1988). Hierarchical Planning involving Deadlines, Travel Time, and Resources. *Computational Intelligence*, **4**, 381-398. Also published in Allen, Hendler and Tate

(1990), 369-386 (Chapter 5, paper 6).

Dechter, R., and Michie, D. (1984a). *Induction of Plans*. TIRM 84-006, The Turing Institute, Glasgow, Scotland.

Dechter, R., and Michie, D. (1984b). *Structured Induction of Plans and Programs*. IBM Los Angeles Scientific Center, Los Angeles, CA, USA.

Dekker, S. T., Herik, H. J. van den, and Herschberg, I. S. (1990). Perfect Knowledge Revisited. *Artificial Intelligence*, **43**, 111-123.

Dietterich, T. G. (1986). Learning at the Knowledge Level. *Machine Learning*, **1**, 287-316. Reprinted in Shavlik and Dietterich (1990), 11-25.

Digitalk. (1988). *Smalltalk/V 286 Object-Oriented Programming System: Tutorial and Programming Handbook*. Digitalk Inc., Los Angeles, CA, USA.

Dijkstra, E. W. (1971). Hierarchical Ordering of Sequential Processes. *Acta Informatica*, **1**, 115-138.

Doran, J. E. (1968). Experiments with a Pleasure-Seeking Automaton. In D. Michie (ed.) *Machine Intelligence 3*. Edinburgh University Press, Edinburgh, Scotland, 195-216.

Doran, J. E. (1969). Planning and Generalisation in an Automaton/Environment System. In B. Meltzer and D. Michie (eds) *Machine Intelligence 4*. Edinburgh University Press, Edinburgh, Scotland, 433-454.

Doran, J. E. (1990). *Using Distributed AI to Study the Emergence of Human Social Organisation*. Department of Computer Science Research Memorandum CSM-154, University of Essex, Colchester, England.

Doran, J. E., Carvajal, H., Choo, Y. J., and Li, Y. (1990). The MCS Multi-Agent Systems Testbed: Developments and Experiments. *Proceedings of the International Working Conference on Cooperating Knowledge-Based Systems* (CKBS'90), Keele University, Keele, England, 240-251.

Drummond, M. (1987). A Representation of Action and Belief for Automatic Planning Systems. In Georgeff and Lansky (1987), 189-212.

Drummond, M., and Currie, K. (1987). Exploiting Temporal Coherence in Nonlinear Plan Construction. *Proceedings of the 7th UK Planning SIG workshop*, British Telecom Research Laboratories, Martlesham Heath, England.

Eckhard, F. (1992). High Performance Capillary Electrophoresis in the Microgravity Environment. *Advanced Space Research*, **12**, 247-255.

Eckhard, F. (1995). Personal communication.

Farhoodi, F., Profitt, J., Woodman, P., and Tunnicliffe, A. (1991). An Approach to the Modelling of Functional Organisation. *Proceedings of the 10th UK Planning SIG workshop*, Logica Cambridge Ltd, Cambridge, England.

Feigenbaum, E. A., and McCorduck, P. (1984). *The Fifth Generation: Artificial Intelligence and*

Japan's Computer Challenge to the World. Pan Books, London, England, ISBN 0 330 284703.

Fikes, R. E., and Nilsson, N. J. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, **2**, 189-208.

Fikes, R. E., Hart, P. E., and Nilsson, N. J. (1972). Learning and Executing Generalised Robot Plans. *Artificial Intelligence*, **3**, 251-288.

Fikes, R., Cutkosky, M., Gruber, T., and Van Baalen, J. (1991). *Knowledge Sharing Technology - Project Overview.* Knowledge Systems Laboratory, Stanford University, Palo Alto, CA, USA. (Also available via WWW from URL `http://www-ksl.stanford.edu/knowledge-sharing/papers/kse-overview.html`.)

Findler, N. V. (1979). *Associative Networks.* Academic Press, New York, NY, USA.

Forsyth, R. (ed.) (1989). *Machine Learning: Principles and Techniques.* Chapman and Hall, London, England, ISBN 0 412 30580 1.

Freeman, P., and Wasserman, A. I. (eds) (1984). *Tutorial on Software Design Techniques.* Fourth edition, IEEE Computer Society Press, New York, NY, USA.

Frost, R. A. (1986). *Introduction to Knowledge Base Systems.* William Collins Sons & Co Ltd, London, England, ISBN 0-00-383114-0. Reprinted 1987.

Gaines, B. R., and Boose, J. H. (1990). *Machine Learning and Uncertain Reasoning.* Volume 3, Knowledge-Based Systems series, Academic Press, London, England, ISBN 0-12-273252-9.

Gasser, L., and Huhns, M. N. (eds) (1989a). *Distributed Artificial Intelligence.* Volume II, Pitman, London, England, ISBN 0 273 08810 6.

Gasser, L., and Huhns, M. N. (1989b). Themes in Distributed Artificial Intelligence Research. In Gasser and Huhns (1989a), pp. vii-xv.

Gehani, N. (1982). Specifications: Formal and Informal - A Case Study. *Software Practice and Experience*, **12**, 433-444.

Genesereth, M. P., and Nilsson, N. J. (1987). *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann Publishers Inc., Palo Alto, CA, USA, ISBN 0-934613-31-1.

Georgeff, M. P. (1987). Planning. *Annual Reviews in Computing Science*, **2**, 357-400. Also published in Allen, Hendler and Tate (1990), 5-25.

Georgeff, M. P., and Lansky, A. L. (1987). *Reasoning about Actions and Plans.* Proceedings of the 1986 Workshop, Timberline, OR, USA, Morgan Kaufmann Publishers Inc., Los Altos, CA, USA, ISBN 0-934613-30-3.

Gold, E. M. (1967). Language Identification in the Limit. *Information and Control*, **37**, 302-320.

Grant, T. J. (1978). Project Daedalus: The Computers. *JBIS*, Special supplement on Project Daedalus, S130-S142.

Grant, T. J. (1985). *Potential Application of Artificial Intelligence Techniques to Aircraft Engineering Management in the Royal Air Force*. Defence Fellowship thesis, UK Ministry of Defence.

Grant, T. J. (1991). Integrating Reactive Planning, Plan Generation, and Planning Operator Induction in the Message-Based Architecture. *Proceedings of the 10th UK Planning SIG workshop*, Logica Cambridge Ltd, Cambridge, England.

Grant, T. J. (1992a). Three Versions and Six Applications of the POI Algorithm. *Proceedings of the 11th UK Planning SIG workshop*, Sussex University, Brighton, England.

Grant, T. J. (1992b). A Review of Multi-Agent Techniques, with application to Columbus User Support Organisation. *Future Generation Computer Systems*, **7**, 413-437.

Grant, T. J. (1992c). Integrating Payload Design, Planning and Control in the Dutch Utilisation Centre. *Proceedings of the Second International Symposium on Ground Data Systems for Space Mission Operations*, Pasadena, CA, USA, JPL Publication 93-5, 237-242.

Grant, T. J., Wigmans, M. H., Eckhard, F., and Eenennaam, J. van. (1992). A Re-Useable Simulation for Space Payloads. *Proceedings of the 1992 European Simulation Multiconference*, York, England, 30-34.

Grant, T. J., Herik, H. J. van den, and Hudson, P. T. W. (1994). Which Blocks World is *the* Blocks World? *Proceedings of the 13th UK Planning and Scheduling SIG workshop*, University of Strathclyde, Glasgow, Scotland.

Greenwell, M. (1988). *Knowledge Engineering for Expert Systems*. Ellis Horwood, Chichester, England, ISBN 0-7458-0513-2.

Gruber, T. R. (1992). *Ontolingua: A Mechanism to Support Portable Ontologies*. Version 3.0, final revision June 1992, Knowledge Systems Laboratory, Stanford University, Palo Alto, CA, USA. (Also available via WWW from URL `http://www-ksl.stanford.edu/knowledge-sharing/ontolingua/index.html`.)

Hammond, K. J. (1986). CHEF: A Model of Case-Based Planning. *Proceedings of the 1986 AAAI Conference*, 261-271. Also published in Allen, Hendler and Tate (1990), 655-659.

Hammond, K. J. (1989). *Case-Based Planning: Viewing Planning as a Memory Task*. Academic Press, London, England.

Harel, D. (1979). *First Order Dynamic Logic*. Lecture Notes in Computer Science, vol 68, Springer-Verlag, Berlin, Germany.

Hart, A. (1989). *Knowledge Acquisition for Expert Systems*. Second edition, New Technology Modular Series, Kogan Page, London, England, ISBN 1-85091-830-9.

Haussler, D. (1988). Quantifying Inductive Bias: AI Learning Algorithms and the Valiant Learning Framework. *Artificial Intelligence*, **36**, 177-221.

Haussler, D. (1989). Learning Conjunctive Concepts in Structural Domains. *Machine Learning*, **4**, 7-40.

Hayes, P. J. (1973). The Frame Problem and Related Problems in Artificial Intelligence. In A. Elithorn and D. Jones (eds) *Artificial and Human Thinking*. Jossey-Bass, San Francisco, CA, USA, 45-59.

Hayes-Roth, F., and McDermott, J. (1978). An Interference Matching Technique for Inducing Abstractions. *CACM*, **26**, 401-410.

Hirsh, H. (1989). *Incremental Version-Space Merging: A General Framework for Concept Learning*. PhD dissertation, Stanford University, Stanford, CA, USA.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, USA, ISBN 0-13-153289-8.

Howden, W. E. (1978). A Survey of Dynamic Analysis Methods. In E. Miller and W. E. Howden (eds) (1981). *Software Testing and Validation Techniques*. IEEE Computer Society Press, New York, NY, USA.

Huhns, M. N. (ed.) (1987). *Distributed Artificial Intelligence*. Pitman, London, England, ISBN 0-273-08778-9.

Huhns, M. N., Mukhopadhyay, U., Stephens, L. M., and Bonnell, R. D. (1987). DAI for Document Retrieval: The MINDS Project. In Huhns (1987), 249-283.

Kadie, C. M. (1988). Diffy-S: Learning Robot Operator Schemata from Examples. *Proceedings of the 5th International Conference on Machine Learning*, Morgan Kaufmann Publishers Inc., San Mateo, CA, USA.

Kalkanis, G., and Conroy, G. V. (1991). Principles of Induction and Approaches to Attribute Based Induction. *Knowledge Engineering Review*, **6**, 307-333.

Kleer, J. de, and Williams, B. C. (1987). Diagnosing Multiple Faults. *Artificial Intelligence*, **32**, 97-130.

Kocabas, S. (1991). A Review of Learning. *Knowledge Engineering Review*, **6**, 195-222.

Kodratoff, Y. (1988). *Introduction to Machine Learning*. Pitman Publishing Ltd, London, England, ISBN 0 273 08796 7.

Kokol, P. (1987). Dining Philosophers - An Exercise in Using JSD. *Software Engineering Notes*, **12**, 27-33.

Korf, R. (1985). Macro-operators: A Weak Method for Learning. *Artificial Intelligence*, **26**, 35-77.

Korf, R. (1987). Planning as Search: A Quantitative Approach. *Artificial Intelligence*, **33**, 65-88.

Köhler, W. (1917). *The Mentality of Apes*. Penguin, Harmondsworth, England.

Laird, J. E., Yager, E. S., Hucka, M., and Tuck, C. M. (1991). Robo-Soar: An integration of external interaction, planning, and learning using Soar. *Robotics and Autonomous Systems*, **8**, 113-129. Also published in van de Velde (1993), 113-129.

Lansky, A. L. (1987). A Representation of Parallel Activity based on Events, Structure, and Causality.

In Georgeff and Lansky (1987), 123-159.

Lefkowitz, L. S., and Lesser, V. R. (1990). Knowledge Acquisition as Knowledge Assimilation. In Gaines and Boose (1990), 37-48.

Lenat, D. B. (1976). *AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search*. STAN-CS-76-570, Computer Science Department, Stanford University, Stanford, CA, USA. Reprinted in R. Davis and D. B. Lenat. (1980). *Knowledge-Based Systems in Artificial Intelligence*. McGraw-Hill, New York, NY, USA.

Marciniak, J. J. (ed.) (1994). *Encyclopedia of Software Engineering*. John Wiley and Sons, New York, NY, USA.

Martial, F. von (1992). *Coordinating Plans of Autonomous Agents*. Lecture Notes in AI, volume 610, Springer-Verlag, Berlin, Germany, ISBN 3-540-55615-X.

Matwin, S., and Morin, J. (1989). Learning Procedural Knowledge in the EBG Context. In Segre (1989), 197-199.

McCarthy, J. (1958). Programs with Common Sense. In *Proceedings of the Symposium on the Mechanisation of Thought Processes*, Her Majesty's Stationery Office, London, England, 77-84. Reprinted in E. A. Feigenbaum, and J. A. Feldman (eds) (1963). *Computers and Thought*. McGraw-Hill, New York, NY, USA.

McCarthy, J., and Hayes, P. J. (1969). Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie (eds) *Machine Intelligence 4*. Edinburgh University Press, Edinburgh, Scotland, 463-502.

McGraw, K. L., and Harbison-Briggs, K. (1989). *Knowledge Acquisition: Principles and Guidelines*. Prentice-Hall, Englewood Cliffs, NJ, USA, ISBN 0-13-517095-8.

Mealy, G. H. (1955). A Method of Synthesising Sequential Circuits. *Bell Systems Technical Journal*, **34**, 1045-1079.

Meyer, B. (1985). On Formalism in Specifications. *IEEE Software*, **January**, 6-23.

Michalski, R. S. (1980). Pattern Recognition as Rule-Guided Inductive Inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-2**, 349-361.

Michalski, R. S. (1986). Understanding the Nature of Learning: Issues and Research Directions. In Michalski, Carbonell and Mitchell (1986), 3-25.

Michalski, R. S. (1993). Toward a Unified Theory of Learning: Multistrategy Task-adaptive Learning. In Buchanan and Wilkins (1993), 7-38. Earlier version published in Reports of Machine Learning and Inference Laboratory, MLI-90-1, AIC, GMU, 1991.

Michalski, R. S., and Larson, J. B. (1978). *Selection of Most Representative Training Examples and Incremental Generation of VL_1 Hypotheses: The underlying methodology and the description of the programs ESEL and AQ11*. Technical Report 867, Computer Science Department, University of Illinois, Urbana, IL, USA.

Michalski, R. S., Carbonell, J. G., and Mitchell, T. M. (eds) (1983). *Machine Learning: An Artificial Intelligence Approach.* Tioga Publishing Company, Palo Alto, CA, USA, ISBN 0-935382-05-4.

Michalski, R. S., Carbonell, J. G., and Mitchell, T. M. (eds) (1986). *Machine Learning: An Artificial Intelligence Approach.* Volume II, Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, ISBN 0-934613-00-1.

Minsky, M. (1975). A Framework for Representing Knowledge. In P. H. Winston (ed.) (1975). *The Psychology of Computer Vision.* McGraw-Hill, New York, NY, USA, 211-277.

Minton, S. (1985). Selectively Generalising Plans for Problem-Solving. In *Proceedings of the 1985 International Joint Conference on AI* (IJCAI-85), Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, ISBN 0-934613-02-8, 596-599. Also published in Allen, Hendler and Tate (1990), 651-654.

Minton, S. (ed.) (1993). *Machine Learning Methods for Planning.* Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, ISBN 1-55860-248-8.

Minton, S., and Zweben, M. (1993). Learning, Planning and Scheduling: An Overview. In Minton (1993), 1-29.

Minton, S., Carbonell, J. G., Etzioni, O., Knoblock, C. A., and Kuokka, D. R. (1987). Acquiring Effective Search Control Rules: Explanation-based learning in the PRODIGY system. In P. Langley (ed.) *Proceedings of the Fourth International Workshop on Machine Learning.* Morgan Kaufmann Publishers Inc., Los Altos, CA, USA.

Mitchell, T. M. (1982). Generalisation as Search. *Artificial Intelligence*, **18**, 203-226.

Mitchell, T. M., Utgoff, P. E., and Banerji, R. B. (1983). Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics. In Michalski, Carbonell and Mitchell (1983), 163-190.

Moore, E. F. (1956). Gedanken-experiments on Sequential Machines. *Automata Studies, Annals of Mathematical Sciences*, **34**, 129-153, Princeton University Press, Princeton, NJ, USA.

Muggleton, S. (1990). *Inductive Acquisition of Expert Knowledge.* Turing Institute Press and Addison-Wesley Publishing Company, Wokingham, England, ISBN 0-201-17561-4.

Murray, K. S. (1987). Multiple Convergence: An Approach to Disjunctive Concept Acquisition. In *Proceedings of the 10th International Joint Conference on AI* (IJCAI-87), Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, ISBN 0-934613-43-5.

Neumann, J. von (1966). *Theory of Self-Reproducing Automata.* Edited and completed by A. W. Burks, University of Illinois Press, Urbana, IL, USA.

Nijssen, G. M., and Halpin, T. A. (1989). *Conceptual Schema and Relational Database Design: A Fact-Oriented Approach.* Prentice Hall of Australia Pty, ISBN 0 7248 0151 0.

Nilsson, N. J. (1980). *Principles of Artificial Intelligence.* Tioga Press, Palo Alto, CA, USA, ISBN 3-540-11340-1.

Nilsson, N. J. (1990). Foreword. In Allen, Hendler and Tate (1990), xi-xii.

Nishimura, H. (1980). Descriptively Complete Process Logic. *Acta Informatica*, **14**, 359-369.

Opie, I., and Opie, P. (1959). *The Lore and Language of Schoolchildren*. Oxford University Press, Oxford, England. Also published by Paladin Frogmore, St. Albans, England, 1977.

Partridge, D., and Paap, K. (1988). An Introduction to Learning. *Artificial Intelligence Review*, **2**, 79-101.

Pedersen, G. S. (1994). *Construction of Multimedia Software Systems for Retrieval and Presentation of Information from Heterogeneous Sources*. PhD thesis, Institute of Computer and Systems Sciences, Copenhagen Business School, Copenhagen, Denmark, ISBN 87-593-8050-0.

Porter, B. W., and Kibler, D. F. (1986). Experimental Goal Regression: A Method for Learning Problem-Solving Heuristics. *Machine Learning*, **1**, 249-286.

Puget, J-F. (1989). Learning Invariants from Explanations. In Segre (1989), 200-204.

Ramsay, A., and Barrett, R. (1987). *AI in Practice: Examples in POP-11*. Ellis Horwood Series in Artificial Intelligence, Chichester, England, ISBN 0-7458-0167-6 (student edition).

Reiter, R. (1978). On Closed World Data Bases. In H. Galliare and J. Minker (eds) *Logic and Data Bases*. Plenum Press, New York, NY, USA, 55-76. Also published in M. Ginsberg (ed.) (1987). *Readings in Nonmonotonic Reasoning*. Morgan Kaufmann Publishers Inc., Los Altos, CA, USA, 300-310.

Rich, E., and Knight, K. (1991). *Artificial Intelligence*. Second edition, International Edition, McGraw-Hill Inc., New York, NY, USA, ISBN 0-07-052263-4.

Ringwood, G. A. (1988). PARLOG86 and the Dining Logicians. *CACM*, **31**, 10-25.

Rosenschein, S. J. (1981). Plan Synthesis: A Logical Perspective. *Proceedings of the 7th International Joint Conference on AI* (IJCAI-81), Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, ISBN 0-934613-044-2, 331-337.

Ruby, D., and Kibler, D. (1989). Learning to Plan in Complex Domains. In Segre (1989), 180-182.

Sacerdoti, E. D. (1977). *A Structure for Plans and Behaviour*. AI series, Elsevier Computer Science Library, Elsevier North-Holland, New York, NY, USA, ISBN 0-444-00209-X.

Salvador, M. S. (1978). Sequencing and Scheduling. Chapter I-9 in J. J. Moder, and S. E. Elmaghraby (eds) (1978). *Handbook of Operations Research*. Volume 2: Models and Applications. Van Nostrand Reinhold, New York, NY, USA, ISBN 0-442-24596-3.

Samuel, A. L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, **3**, 211-229. Reprinted in E. A. Feigenbaum and J. A. Feldman (eds) (1963). *Computers and Thought*. McGraw-Hill, New York, NY, USA, 71-105.

Searle, J. R. (1980). Minds, Brains, and Programs. *The Behavioural and Brain Sciences*, **3**, 417-424. Reprinted in D. R. Hofstadter and D. C. Dennett (eds) (1981). *The Mind's I*. Penguin Books, Harmondsworth, England, 252-373, and in Boden (1990), 67-88.

Segre, A. M. (ed.) (1989). *Proceedings of the 6th International Workshop on Machine Learning (ML'89)*. Morgan Kaufmann Publishers Inc., San Mateo, CA, USA.

Segre, A. M. (1991). Learning How to Plan. *Robotics and Autonomous Systems*, **8**, 93-111. Also published in van de Velde (1993), 93-111.

Shavlik, J. W. (1989). An Empirical Analysis of EBL Approaches for Learning Plan Schemata. In Segre (1989), 183-187.

Shavlik, J. W., and Dietterich, T. G. (1990). *Readings in Machine Learning*. Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, ISBN 1-55860-143-0.

Shaw, M. J., and Whinston, A. B. (1989). Learning and Adaptation in Distributed Artificial Intelligence Systems. In Gasser and Huhns (1989a), 413-429.

Shlaer, S., and Mellor, S. J. (1992). *Object Lifecycles: Modeling the World in States*. Yourdon Press computing series, Prentice Hall, Inc., Englewood Cliffs, NJ, USA, ISBN 0-13-629940-7.

Shoham, Y. (1986). Temporal Reasoning. In S. C. Shapiro (ed.) *Encyclopaedia of Artificial Intelligence*. John Wiley and Sons, New York, NY, USA.

Shoham, Y. (1987). What is the Frame Problem? In Georgeff and Lansky (1987), 83-98.

Sian, S. S. (1991). The Role of Cooperation in Multi-Agent Learning. In S. M. Deen (ed.) (1991). *Proceedings of the 1990 International Working Conference on Cooperating Knowledge Based Systems* (CKBS'90). Springer-Verlag, London, England, ISBN 3-540-19649-8, 164-180.

Simon, H. A. (1983a). Why Should Machines Learn? In Michalski, Carbonell and Mitchell (1983), 25-37.

Simon, H. A. (1983b). Search and Reasoning in Problem-Solving. *Artificial Intelligence*, **21**, 7-29.

Simon, H. A., and Lea, G. (1974). Problem Solving and Rule Induction: A unified view. In L. Gregg (ed.) *Knowledge and Cognition*. Lawrence Erlbaum, Hillsdale, NJ, USA, 105-127.

Smith, J. M., and Smith, D. C. P. (1977). Database Abstractions: Aggregation and Generalisation. *ACM Transactions in Database Systems*, **2**, 105-133.

Sowa, J. F. (1984). *Conceptual Structures*. Addison-Wesley, Reading, MA, USA, ISBN 0-201-14472-7.

Sparck Jones, K. (1990). Semantic Networks. In A. Bundy (ed.) (1990). *The Catalogue of Artificial Intelligence Techniques*. Third edition, Springer-Verlag, Berlin, Germany, 120-121.

Steel, S. (1987). The Bread and Butter of Planning. *Artificial Intelligence Review*, **1**, 159-181.

Steel, S. (1988). Topics in Planning. In R. T. Nossum (ed.) *Advanced Topics in Artificial Intelligence*. Second Advanced Course, ACAI'87, Oslo, Norway, Lecture Notes in Artificial Intelligence, Volume 345, Springer-Verlag, Berlin, 146-188, ISBN 3-540-50676-4.

Steels, L. (1989). *Cooperation between Distributed Agents through Self-Organisation*. AI memo 89-5.

Vrije Universiteit, Brussels, Belgium.

Stefik, M. J. (1981). Planning with Constraints. *Artificial Intelligence*, **16**, 111-140. Also published in Allen, Hendler and Tate (1990), 171-185 (Chapter 3, paper 6).

Stuart, C. J. (1985). *Synchronisation of Multiagent Plans using a Temporal Logic Theorem Prover.* Technical Note 350, Stanford Research Institute, Menlo Park, CA, USA.

Suchman, L. A. (1987). *Plans and Situated Actions: The problem of human-machine communication.* Cambridge University Press, Cambridge, England, ISBN 0 521 33739 9.

Sussman, G. J. (1973). *A Computational Model of Skill Acquisition.* Technical Report 297, AI Laboratory, MIT, Cambridge, MA, USA.

Sussman, G. J. (1975). *A Computer Model of Skill Acquisition.* Elsevier, New York, NY, USA.

Sutton, R. S. (1990). Integrated Architectures for Learning, Planning, and Reacting based on Approximating Dynamic Programming. In *Proceedings of the 7th International Conference on Machine Learning*, Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, 216-224.

Tate, A. (1975). *Using Goal Structure to Direct Search in a Problem Solver.* PhD thesis, MIRU, Edinburgh University, Edinburgh, Scotland.

Tate, A. (1976). *Project Planning Using a Non-Linear Planner.* DAI Research Report no. 25, Department of Artificial Intelligence, Edinburgh University, Edinburgh, Scotland.

Tate, A., Hendler, J., and Drummond, M. (1990). A Review of AI Planning Techniques. In Allen, Hendler, and Tate (1990), 26-49.

Tenenberg, J. D. (1991). Abstraction in Planning. In Allen, Kautz, Pelavin and Tenenberg (1991), Chapter 4, 213-283.

Thornton, C., and Boulay, B. du. (1992). *Artificial Intelligence through Search.* Intellect Books, Oxford, England, ISBN 1-871516-24-2.

Tomlinson, C., Scheevel, M., and Won Kim. (1989). Sharing and Organisation Protocols in Object-Oriented Systems. *Journal of Object-Oriented Programming*, **2**, 25-36.

Turing, A. M. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, **42**, 230-265. Corrected *ibid.*, **43**, 544-546 (1937).

Turner, R. (1984). *Logics for Artificial Intelligence.* Ellis Horwood Series Artificial Intelligence, Chichester, England, ISBN 0-85312-713-1.

Valente, A. (1994). Planning. In Breuker and van de Velde (1994), chapter 10, 213-230.

Valente, A. (1995). Knowledge-Level Analysis of Planning Systems. *SIGART Bulletin*, **6**.

Velde, W. van de (ed.) (1993). *Towards Learning Robots.* MIT Press, Cambridge, MA, USA, ISBN 0-262-72017-5.

Vere, S. A. (1978). Inductive Learning of Relational Productions. In Waterman and Hayes-Roth (1978), 281-295.

Waterman, D. A. (1970). Generalization Learning Techniques for Automating the Learning of Heuristics. *Artificial Intelligence*, **1**, 121-170.

Waterman, D. A., and Hayes-Roth. F. (eds) (1978). *Pattern-Directed Inference Systems*. Academic Press, New York, NY, USA.

Wielinga, B. J., Schreiber, A. Th., and Breuker, J. A. (1992). KADS: A Modelling Approach to Knowledge Engineering. *Knowledge Acquisition*, **4**, 5-53. Also published in Buchanan and Wilkins (1993), 92-116.

Wilensky, R. (1983). *Planning and Understanding: A Computational Approach to Human Reasoning*. Addison-Wesley, Reading, MA, USA, ISBN 0-201-09590-4.

Wilkins, D. E. (1988). *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, ISBN 0-934613-94-X.

Winograd, T. (1972). *Understanding Natural Language*. Academic Press, New York, NY, USA.

Winston, P. H. (1979). *Artificial Intelligence*. Addison-Wesley, Reading, MA, USA, second printing, ISBN 0-201-08454-6.

Winston, P. H. (1984). *Artificial Intelligence*. Second edition, Addison-Wesley, Reading, MA, USA, reprinted with corrections July 1984, ISBN 0-201-08259-4.

Winston, P. H. (1993). *Artificial Intelligence*. Third edition (1992), reprinted with corrections May 1993, Addison-Wesley, Reading, MA, USA, ISBN 0-201-53377-4.

Zweben, M., Davis, E., Daun, B., Draschler, E., Deale, M., and Eskey, M. (1993). Learning to Improve Constraint-Based Scheduling. *Artificial Intelligence*, **58**, 1-3.

Summary

This thesis addresses the research question:

"Where do planning operators come from?"

Planning operators are data-structures that represent some knowledge about classes of possible actions in a problem domain. They are used in *plan generation*, which is defined as the process of selecting and instantiating actions from a set of planning operators and logically ordering these instantiated actions in a sequence that will, on execution, transform a given initial domain state into a desired ("goal") state.

Until now, planning system developers have been responsible for providing suitable planning operators for a given problem domain, in the same way as knowledge engineers have been responsible for providing suitable production-rules in expert systems. Before domain-specific knowledge can be provided, it must be acquired. In the expert-systems field it has been known since the early 1980s that *knowledge acquisition* is a difficult and laborious process. There has been progress in automating the formulation of production-rules, especially by inducing production-rules from examples. In the non-AI discipline of *software engineering*, the laborious and error-prone nature of the analogous process of *requirements analysis* has been recognised for at least three decades. Recently, similar difficulties have been acknowledged to exist in the compilation of planning knowledge (Minton and Zweben, 1993). The analogy with the induction of production-rules suggests the possibility of inducing planning operators from examples. This possibility motivated my research.

In this thesis, I claim that planning operators can be induced *ab initio* from a domain model represented as collections of domain objects, inter-object relationships, and interrelationship constraints. Moreover, such a domain model can be extracted from a collection of state-descriptions.

I substantiate this claim by developing an induction algorithm - known as the *Planning Operator Induction* (POI) algorithm - by implementing the algorithm, and by performing various experiments on the implemented programs. To do this, I had to develop:

- *the POI ontology.* An *ontology* is a set of definitions of content-specific knowledge-representation primitives that is both human and machine readable (Gruber, 1992). The POI ontology is based on primitives found in the *entity-relationship model* (Chen, 1976) of domain structure and in the *state-transition model* of domain change (Shlaer and Mellor, 1992).

- *a family of meta-heuristics.* A *meta-heuristic* is a form of control structure that applies to a set of domains (Sowa, 1984), i.e., it is domain-independent.

My research differs from other approaches to learning planning knowledge in that it:

- *learns domain knowledge.* Most of the research to date has concentrated on learning control knowledge, i.e., knowledge indicating how the planning system should control its search for a plan during plan generation.

- *induces behavioural knowledge from structural knowledge.* Other approaches have been restricted to behavioural knowledge alone.

- *represents the induced knowledge as planning operators.* A variety of knowledge representations are used in other induction systems, e.g., production rules.

- *uses an ontology grounded on mature software engineering principles.* Other researchers have largely failed to document their ontologies.

- *enables the induction of behavioural domain knowledge ab initio.* Most research has been concerned with the easier task of automating knowledge refinement (Carbonell and Gil, 1990).

- *does not make use of any control knowledge.* Other algorithms depend, at least in part, on inputs that provide domain-specific information on the sequencing of states and/or actions.

There are six chapters in this thesis. Additional material includes a references section, an index, a summary (in English and in Dutch), and my curriculum vitae.

Chapter 1 opens with the motivation for my research, states my thesis claim, and places my research in context. Knowledge-based planning, software engineering, machine learning, and multi-agent systems techniques are identified as related fields. The POI algorithm is outlined and distinguished from plan generation. Potential applications of POI are as a knowledge acquisition tool for planners, as an element of multi-agent systems, as a Computer Aided Software Engineering tool, as a commercial or military intelligence support tool, and as an instruction support tool for teachers. The POI algorithm is limited in that it does not guarantee perfection, efficiency, optimality, or applicability to all problems and domains. Aspects of the related fields which are specifically excluded from my research are:

- scheduling, design, or other planning specialisations involving metric quantities;

- uncertainty arising from forgetful agents, from imprecise, incorrect, or conflicting domain knowledge, or from incompleteness in initial or goal state descriptions;

- the selection of series of observations;

- the recognition of objects and their classes in world-state observations.

The extensive use of software engineering ideas and methods is emphasised in taking an engineering approach to my research. The history of my research is summarised. Finally, the thesis conventions and layout are stated.

Chapter 2 reviews the relevant literature in the related fields. Software engineering and knowledge-based planning were the sources of the POI ontology. The types of knowledge that need to be represented in the behavioural part of the ontology were identified from Tate, Hendler and Drummond's (1990) definition of plan generation. The structural part was based on Chen's (1976) entity-relationship model, enhanced by means of Nijssen and Halpin's (1989) exclusion constraints. The aspects of the POI ontology that are unusual from the knowledge-based planning viewpoint are:

- the provision of a structural domain model;

- the description of states in terms of inter-object relations;

- the representation of plans as sequences of states, rather than sequences of transitions (i.e.,

instantiated operators); and

- the emphasis on representing domain constraints.

Machine learning is the source of the POI's core inferencing process. The POI algorithm encapsulates an inductive technique known as the *version space and candidate elimination* algorithm (Mitchell, 1982). Other learning systems in the planning area are surveyed, showing that they either learn control knowledge or make use of sequencing information. Multi-agent systems techniques were used in *closed-loop* testing of the POI algorithm. Single- and multi-agent learning are contrasted, and the role of inter-agent cooperation in learning is summarised.

Chapter 3 documents the POI algorithm and its underlying ontology. Functionality, application, and tractability requirements are compiled. The POI ontology connects a structural model of the problem domain to its behavioural model by means of a linking model. The ontology is presented using Chen's (1976) entity-relationship notation. Since the POI algorithm also takes its input in the form of an entity-relationship model of the domain, the POI ontology is a *meta-representation.*

Following accepted software engineering practice, the POI algorithm is described in terms of its top-down, functional decomposition. At the top level, the algorithm is decomposed into two Parts. At the second level of decomposition, the algorithm consists of nine steps. Each step is described in detail, with some steps being decomposed to a third level. Single- and multi-agent implementations of the POI algorithm are described, with the implementation-language classes being related to the entity-classes in the POI ontology. The behaviour of the single-agent program is shown as a state-transition network. Key refinements made during implementation are outlined.

The limitations of the POI algorithm are described. The most serious limitation is that the algorithm is inherently combinatorially explosive. The algorithm's complexity is analysed theoretically, showing that the worst-case runtime and memory usage would be proportional to 2 to the power of the number of sentences in the state-description language. Potential countermeasures to the combinatorial explosion are described, including the use of domain-specific heuristics, a plan-space lattice, a class-level version space, inheritance, whole-part decomposition, and version-space partitioning. The inheritance and version-space partitioning countermeasures have been implemented.

Chapter 4 describes a series of *open-loop* experiments designed to:

- demonstrate that the POI algorithm was capable of inducing planning operators;

- investigate the algorithm's complexity empirically.

Both programme goals have been achieved. Using the single-agent program, sets of planning operators have been induced for eight domains ranging from one to 24 object-classes. Extracts of the outputs for three selected domains have been documented. Planning-operator sets to be found in the AI literature can be reproduced. Furthermore, novel planning operators can be induced, as shown using the High Performance Capillary Electrophoresis domain. Several of these novel operators could not have been learned by other operator-learning algorithms.

The sensitivity of the POI algorithm has been investigated by varying the numbers of object-classes, object-instances, relationship-classes, and domain constraints, and by applying each of the four meta-heuristics in turn. The results of these investigations are summarised. The complexity of the POI algorithm is investigated empirically, both within a given domain and across several domains. Within-domain complexity was found to be consistent with exponential behaviour, and across-domain

complexity is consistent with a third-order polynomial.

The effectiveness of the countermeasures was compared with the baseline algorithm. For small numbers of object-classes and of object-instances per class, the introduction of inheritance was found to be more effective than version-space partitioning. Combining the countermeasures is more effective than either countermeasure alone. The introduction of inheritance has also been shown to be effective in enabling domain constraints to be modelled completely using binary exclusion-rules. Finally, the fact that the POI ontology is a meta-representation was exploited to perform *meta-runs*, i.e., POI runs for which POI itself was the problem (meta-)domain.

Chapter 5 describes a second series of experiments designed to close the loop in testing the POI algorithm. The loop was closed by embedding the full POI algorithm, together with reactive and deliberative planning functionalities, in a *POIAgent*. One or more POIAgents were placed in an environment consisting of several other simpler agents, which simulated the POIAgents' problem domain. At the start of each run, the POIAgents were *naive*, i.e., they had no prior domain knowledge. Each POIAgent was given a series of goals to achieve by interacting with the simpler agents in its environment. The goal-series were designed so that the POIAgents acquired knowledge about their environment, induced a set of planning operators from the acquired knowledge, generated a plan using the induced operators, and executed that plan successfully. In short, the POIAgents performed *learning-by-doing* (Anzai and Simon, 1979).

Some experiments were done with a single POIAgent, to investigate single-agent learning in a multi-agent environment. Using the blocks world (Winograd, 1972), I showed that, given a selected pair of 3-blocks-world states, a POIAgent induced correctly the complete set of planning operators from Nilsson (1980). Other experiments were done with two POIAgents, to investigate multi-agent learning. Neither POIAgent gained complete knowledge of the overall domain by its own observations. They had to exchange information they had acquired about their individual problem domains. The information exchanged was represented as a domain model. To perform *learning-by-being-told*, recipient POIAgents had to be given additional functionality to *assimilate* (Lefkowitz and Lesser, 1990) the received domain models with their own domain model.

Chapter 6 summarises the results of my research. It concludes that it is feasible to:

- automate the learning of planning operators.
- induce planning operators *ab initio* from collections of domain objects, inter-object relationships, and interrelationship constraints, as claimed.
- compile such lists from unordered observations of non-adjacent domain states.
- induce planning operators for complex, real-world domains.

The contributions of my research are summarised for each of the related fields. Directions for future research are indicated. Finally, the significance of my research, both in AI and in software engineering, is identified. For AI, the research makes it possible to generate automatically sets of planning operators consistent with the structural model of a domain. For software engineering, the research makes it possible to generate a state-transition network from an entity-relationship model. This could both save effort in software design and reduce errors arising from inconsistency between the structural and behavioural models.

Samenvatting

Titel in het Nederlands:

Inductief leren van op kennis gebaseerde planningsoperatoren

Dit proefschrift beschrijft onderzoek naar de vraag:

"Waar komen planningsoperatoren vandaan?"

Planningsoperatoren zijn datastructuren die kennis representeren over klassen van mogelijke acties in een toepassingsdomein. Ze worden gebruikt in het planningsproces. *Op kennis gebaseerde planning* is gedefineerd als het proces van selecteren en instantiëren van acties vanuit een verzameling planningsoperatoren, om vervolgens de geïnstantieerde acties in een logische volgorde te zetten. Na uitvoering van de acties is een initiële domeintoestand omgezet in een gewenste ("doel") toestand.

Tot nu toe zijn de ontwikkelaars van planningssystemen verantwoordelijk geweest voor het ontwerp van de juiste planningsoperatoren voor een bepaald toepassingsdomein, en wel op dezelfde manier als kennistechnologen verantwoordelijk zijn voor relevante regels in expertsystemen. Voordat relevante domein-specifieke kennis geleverd kan geworden, moet zij eerst verworven worden. Binnen het gebied van expertsystemen weten we sinds het begin van de tachtiger jaren dat het verwerven van kennis een moeilijk en bewerkelijk proces is. Wel is er vooruitgang geboekt met het automatiseren van het formuleren van productieregels, met name in de inductie van productieregels aan de hand van voorbeelden. Buiten de AI staat een vergelijkbaar proces van analyse van gebruikerseisen bekend als bewerkelijk en foutgevoelig. In de laatste jaren is duidelijk geworden dat binnen de AI overeenkomstige moeilijkheden bestaan bij het verwerven van planningskennis (Minton en Zweben, 1993). De analogie van planningssystemen met expertsystemen suggereert dat het goed mogelijk is om planningsoperatoren vanuit voorbeelden te induceren. Deze mogelijkheid was de motivatie voor mijn onderzoek.

In dit proefschrift beweer ik dat het mogelijk is om planningsoperatoren te induceren vanaf het begin vanuit een domeinstructuurmodel dat wordt gerepresenteerd door verzamelingen van domein-objecten, relaties tussen objecten, en constraints tussen relaties. Zulk een domeinstructuurmodel kan worden verkregen vanuit een verzameling toestandsbeschrijvingen.

Om mijn bewering te onderbouwen heb ik een inductie-algoritme ontwikkeld: het *Planning Operator Induction* (POI) algoritme. Het POI-algoritme is geïmplementeerd en er zijn verscheidene experimenten mee uitgevoerd. Voor de ontwikkeling van het algoritme heb ik tevens ontworpen:

- *De POI-ontologie.* Een *ontologie* is een verzameling definities van kennisrepresentatie-primitieven die zowel voor mensen als voor machines leesbaar is (Gruber, 1992). De POI-ontologie is voor de domeinstructuur gebaseerd op primitieven van Chen's (1976) *entiteit-relatie model* en voor het domeingedrag op primitieven van het *toestandsmodel* (Shlaer en Mellor, 1992).

- *Een familie van meta-heuristieken.* Een *meta-heuristiek* is een besturingsstructuur die toepasbaar is op een verzameling domeinen (Sowa, 1984), dat wil zeggen de meta-heuristiek is domeinonafhanklijk.

Mijn onderzoek wordt gekenmerkt door de volgende verschillen met andere aanpakken voor het leren

van planningskennis, doordat het:

- *domeinkennis leert.* De meerderheid van onderzoek tot nu toe heeft zich gericht op het leren van besturingskennis, dat wil zeggen kennis die tijdens het planningsproces het zoekproces bestuurt.

- *gedragskennis van structuurkennis induceert.* Ander onderzoek is beperkt tot alleen gedragskennis.

- *de geïnduceerde kennis als planningsoperatoren representeert.* Er zijn verschillende representaties in gebruik in andere inductieve systemen, bijvoorbeeld productie-regels.

- *een ontologie gebruikt die gebaseerd is op voldragen software-engineering principes.* In het algemeen hebben andere onderzoekers zijn ontologieën niet gedocumenteerd.

- *induceren van gedragskennis mogelijk maakt vanaf het begin.* Het meeste onderzoek betreft de minder moeilijke taak van het automatiseren van kennisverfijning (Carbonell en Gil, 1990).

- *geen gebruik van besturingskennis maakt.* Andere algoritmen zijn afhankelijk, gedeeltelijk of in zijn geheel, van invoer-informatie die domein-specifieke kennis geeft over de volgorde van toestanden en/of acties.

Dit proefschrift bestaat uit zes hoofdstukken. Bijgevoegd materiaal bestaat uit referenties, samenvattingen in het Engels en Nederlands, mijn curriculum vitae, en een trefwoordenregister.

Hoofdstuk 1 formuleert de motivatie en probleemstelling van deze studie en plaatst het onderzoek in zijn context. Op kennis gebaseerde planning, software-engineering, lerende systemen en multi-agent systemen worden als gerelateerde gebieden geïdentificeerd. Het POI-algoritme wordt beschreven en de verschillen met planning worden duidelijk gemaakt. Als potentiële toepassingen noemen we een kennisacquisitie-hulpmiddel voor planners, een functioneel element van multi-agent systemen, een Computer Aided Software Engineering hulpmiddel, een ondersteunend hulpmiddel voor commerciële of militaire inlichtingen, en een ondersteunend hulpmiddel voor leraren. Het POI-algoritme is beperkt in die zin dat het geen garantie biedt voor volledigheid, efficiency, optimaliteit, of toepasbaarheid voor alle mogelijke problemen en domeinen. Aspecten van de gerelateerde gebieden die niet voorkomen in mijn onderzoek zijn:

- scheduling, ontwerpen, of andere specialisaties van planning met metrische eenheden;

- onzekerheid die afkomstig is van agenten die kunnen vergeten, van domeinkennis die onnauwkeurig, incorrect of conflicterend is, of van initiële- of doelstandstanden die onvolledig zijn;

- het selecteren van een reeks toestandswaarnemingen;

- het herkennen van objecten en hun klassen in de toestandswaarnemingen.

Mijn aanpak, die sterk gebaseerd is op begrippen en methoden uit de software-engineering, wordt precies uitgelegd. De geschiedenis van mijn onderzoek wordt geschetst. Conventies en structuur van het proefschrift worden als leidraad gegeven.

Hoofdstuk 2 laat de relevante literatuur in de gerelateerde gebieden de revue passeren. Software-engineering en op kennis gebaseerde planning zijn de bronnen voor de POI-ontologie. De typen van domeingedragkennis zijn ontleend aan Tate, Hendler en Drummond's (1990) definitie van planning. De typen van domeinstructuurkennis zijn gebaseerd op Chen's (1976) entiteit-relatie model, uitgebreid met Nijssen en Halpin's (1989) uitsluitingsconstraints. Aspecten van de POI-ontologie die ongewoon zijn vanuit het oogpunt van op kennisgebaseerde planning zijn:

- het leveren van een domeinstructuurmodel;
- de beschrijving van domeintoestanden in termen van relaties tussen domeinobjecten;
- de representatie van plannen als een reeks van toestanden, in plaats van een reeks van acties (dat wil zeggen geïnstantieerde operatoren); en
- de nadruk op het representeren van domeinconstraints.

De *version space and candidate elimination* techniek van lerende systemen (Mitchell, 1982) is de kern van het POI-algoritme. Andere technieken bij lerende systemen in het planningsgebied gebruiken òf besturingskennis òf kennis over de volgorde van de operatoren. Multi-agent systeemtechnieken worden gebruikt in het *closed-loop* testen van het POI-algoritme. Het contrast tussen enkel- en multi-agent leren wordt aangegeven, en de rol van inter-agent samenwerking in het leerproces wordt samengevat.

Hoofdstuk 3 documenteert het POI-algoritme en de POI-ontologie. Functionele, toepassings-, en handelbaarheidseisen worden bijeengebracht. De POI-ontologie verbindt het domeinstructuurmodel met het domeingedragmodel door middel van een verbindingsmodel. Chen's (1976) entiteit-relatie model wordt gebruikt om de POI-ontologie te representeren. Omdat het POI-algoritme een entiteit-relatie model van een domein als invoer neemt is de POI-ontologie een *meta-representatie*.

Het POI-algoritme is ontworpen door middel van een bekende software-engineering techniek, namelijk de top-down, functionele decompositie. Op de hoogste niveau zijn er twee delen en op de tweede niveau zijn er negen stappen. Sommige stappen zijn te ontleden tot op een derde niveau. Elke stap in het algoritme wordt in detail beschreven. Twee implementaties worden gepresenteerd, namelijk de single-agent implementatie en de multi-agent implementatie. De relatie tussen klassen in de POI-ontologie en klassen in de implementatietaal wordt daarbij uitgelegd. Het gedrag van de single-agent implementatie wordt beschreven door middel van een toestandsdiagram. Verfijningen gemaakt tijdens de implementatie worden geschetst.

De beperkingen van het POI-algoritme worden geïdentificeerd. Het belangrijkste hierbij is dat het algoritme combinatorisch-explosief is. De complexiteit van het algoritme wordt theoretisch geanalyseerd, met als resultaat dat, in het ergste geval, tijd en geheugen proportioneel zijn met twee tot de macht van het aantal zinnen in de domeinbeschrijvingstaal. Diverse tegenmaatregelen worden geformuleerd, waaronder het gebruik van domein-specifieke heuristieken, van roosters in planningsruimten, van version space op het niveau van een klasse, van overerving, van decompositie, en van version-space verdeling. Twee tegenmaatregelen zijn geïmplementeerd, namelijk overerving en version-space verdeling.

Hoofdstuk 4 beschrijft een serie van *open-loop* experimenten. Deze experimenten zijn ontworpen om twee doelen te bereiken:

- te demonstreren dat het POI-algoritme planningsoperatoren kan induceren;

- de werkelijke complexiteit te onderzoeken.

Beide doelen worden gehaald. Planningsoperatoren zijn voor acht domeinen geïnduceerd, met een omvang variërend van 1 tot 24 object-klassen. Delen van de uitvoer voor drie geselecteerde domeinen zijn in dit proefschrift gedocumenteerd. Planningsoperatoren uit de AI-literatuur kunnen gereproduceerd worden. Verder kunnen operatoren geïnduceerd worden die nooit eerder zijn beschreven, zoals wordt aangetoond met behulp van het High Performance Capillary Electrophoresis domein. Verschillende van die nooit-eerder-geïnduceerde operatoren kunnen andere operator-lerende algoritmen niet genereren.

Door het aantal object-klassen, object-instanties, relatie-klassen, en domein-constraints te variëren, en door elke meta-heuristiek toe te passen bestuderen we de gevoeligheid van het POI-algoritme. De resultaten zijn samengevat. De werklijke complexiteit, zowel binnen een bepaald domein als tussen domeinen, is gemeten. De complexiteit binnen een domein komt overeen met exponentieel gedrag, en de complexiteit tussen de domeinen komt overeen met een polynoom van de derde orde.

De effectiviteit van de tegenmaatregelen wordt vergeleken met de effectiviteit van het baseline-algoritme. Voor een beperkt aantal object-klassen en object-instanties per klasse is het introduceren van overerving effectiever dan version-space verdeling. Een combinatie van tegenmaatregelen blijkt effectiever dan één maatregel alleen. Overerving maakt het ook mogelijk om alle domein-constraints te representeren door middel van binaire uitsluitingsregels. Tenslotte wordt het feit dat de POI-ontologie een meta-representatie is, uitgebuit door experimenten te doen waarin POI tevens het toepassingsdomein is.

Hoofdstuk 5 beschrijft een serie van *closed-loop* experimenten. Het POI-algoritme wordt daarvoor omgeven door toegevoegde functionaliteiten voor reactieve en generatieve planning in een zogenaamde *POIAgent*. Eén of twee POIAgenten worden geplaatst in een omgeving met meerdere, eenvoudige agenten, die het POIAgentendomein nabootsen. Elke POIAgent is in het begin naïef, dat wil zeggen zij heeft geen kennis over het domein. Iedere POIAgent is voorzien van een serie taken die uitgevoerd moet worden door interactie met de eenvoudige agenten. De taken zijn zo ontworpen dat de POIAgenten kennis over het domein verwerven, operatoren induceren, plannen genereren met die operatoren, en die plannen uitvoeren. In het kort, de POIAgenten presteren volgens *learning-by-doing* (Anzai en Simon, 1979).

Sommige experimenten worden met één POIAgent gedaan. Met behulp van het domein van de blocks world (Winograd, 1972) toon ik aan dat een POIAgent de volledige verzameling planningsoperatoren, zoals door Nilsson (1980) gedefinieerd is, kan induceren. Andere experimenten worden met twee POIAgenten gedaan. Geen van beide POIAgenten kon hierbij afzonderlijk de volledig kennis vergaren, maar elk van hen moest informatie uitwisselen om tot een volledige verzameling planningsoperatoren te komen. De informatie wordt als een domeinstructuurmodel uitgewisseld. Om *learning-by-being-told* te implementeren is het nodig dat aan elke POIAgent een functionaliteit wordt toegevoegd, zodat een POIAgent het domeinstructuurmodel met zijn eigen domeinstructuurmodel kan *assimileren* (Lefkowitz en Lesser, 1990).

Hoofdstuk 6 vat de resultaten van mijn onderzoek samen. De belangrijke conclusies zijn:

- het leren van planningsoperatoren kan geautomatiseerd worden;

- planningsoperatoren kunnen inderdaad, zoals ik beweerd heb, geïnduceerd worden vanaf het begin vanuit verzamelingen domein-objecten, relaties tussen objecten, en constraints tussen relaties;

- soortgelijke domeinstructuurmodellen kunnen verkregen worden vanuit een verzameling toestandsbeschrijvingen;

- planningsoperatoren kunnen geïnduceerd worden voor complexe praktijkdomeinen.

De bijdrage van mijn onderzoek tot ieder afzonderlijk gerelateerd gebied wordt samengevat. Indicaties worden gegeven voor mogelijk toekomstig onderzoek door anderen. Tenslotte wordt de betekenis van mijn onderzoek geïdentificeerd voor zowel de AI als voor de software-engineering. Voor de AI maakt mijn onderzoek het mogelijk om planningsoperatoren consistent met een domeinstructuurmodel automatisch te genereren. Voor de software-engineering maakt mijn onderzoek het mogelijk om een toestandsdiagram automatisch te genereren vanuit een entiteit-relatie model. Dit kan zowel inspanning in het ontwerpen van software besparen, als een bron van menselijke fouten elimineren.

Curriculum Vitae

Timothy John Grant was born in Stokes Valley, New Zealand, on 18 June 1947. He attended schools in England and Kenya, gaining five Advanced and one Special level General Certificates of Education from Tonbridge School, England, in 1966. While still at school, he obtained a Private Pilot's Licence.

From 1966 to 1969, he studied Aeronautical Engineering at the University of Bristol, England, sponsored by the Royal Air Force. He graduated with a Bachelor of Science degree and 172 flying hours with the University's Air Squadron.

After officer training in 1969 to 1970, he filled a variety of posts in the Engineering Branch of the Royal Air Force. In his first appointment from 1970 to 1972, he commanded 80 military technicians responsible for the maintenance of 24 Hercules transport aircraft. From 1972 to 1974, he commanded 75 civilian and military personnel with varying nationalities, races and religions in Singapore, with responsibilities also covering Royal Air Force aircraft in Bangkok, Djakarta, and Saigon. Over the following three years, he was in charge of a department in deepest Norfolk, responsible for calculating the purchasing requirements for spare parts for all Royal Air Force and Royal Navy aircraft. During this period he developed a program for performing the spares purchasing calculations; this was recognised by the award of the Queen's Silver Jubilee medal in 1977. From 1977 to 1980 he commanded a squadron which supported the operations of eight Vulcan bomber aircraft. He was appointed to the Chief Scientist's (Royal Air Force) department from 1980 to 1983, developing a suite of computer programs for simulating wartime operations. One of the programs was used in earnest during the 1982 Falklands War. In 1983, he was awarded a Defence Fellowship to survey during a sabbatical year at Brunel University, England, the potential applications of AI to aircraft maintenance management in the Royal Air Force. His last appointment was to the UK Ministry of Defence department responsible for policy for computing support for aircraft maintenance management, where he oversaw 200 projects, of which five were AI-based. He retired from the Royal Air Force in the rank of Squadron Leader (equivalent to Major) in September 1987.

On leaving the Royal Air Force, he emigrated to The Netherlands, taking up employment as a Senior Consultant with a leading Dutch software house (Origin/BSO). He specialised in the space market, and has been involved in a large variety of software development projects in capacities ranging from project definition, acquisition, and management, as well as requirements analysis, architectural design, and prototyping. In addition to the DUC-ASS mentioned in this thesis, notable projects have included: an expert system to support astronauts in monitoring and controlling safety-critical spacecraft subsystems (such as the Cabin Atmospheric Subsystem); an object-oriented simulator of the High Performance Capillary Electrophoresis spacecraft payload; the knowledge-based planning module embedded within the European Space Software Development Environment; and the user-interface software for the Russian part of the International Space Station.

Tim Grant is married, with two daughters. The whole family enjoys music, especially piano-playing.

Index