Integration of Classical and Model-Based Technologies for the Automated Synthesis of Plans

Peter A Jarvis

A thesis submitted in partial fulfilment of the requirements of the University of Brighton for the degree of

Doctor of Philosophy

School of Computing and Mathematical Sciences
The University of Brighton
In collaboration with
The Llewellyn Group of Companies

August 1997

Abstract

- **Context and Objective:** Whilst offering a number of desirable features, classical planners have yet to achieve wide spread application to "industrial" applications. Model-Based Planners, in contrast, have been successfully applied to a number of "industrial" problems. This thesis examines both technologies to justify, design, and evaluate an integrated architecture that exploits their relative strengths.

- **Research Infrastructure:** To provide the infrastructure for considering an integrated architecture, a KADS model of construction industry planning knowledge and a classical planning workbench are developed. The workbench is built around a principled object oriented design derived from the O-Plan system. This effort permits experimentation with classical planner components and provides the classical element of the final integrated architecture. The KADS model set is constructed through a combination of interviews with practitioners and observations at a construction site. The set details the role of planning within the construction process and the domain expertise applied in the planning task. This understanding of "industrial" application requirements is used to identify the complementary strengths between classical and model based planning and evaluate the integrated architecture.

- **Complementary Strengths:** Encoding the KADS model of construction expertise and enhancements to existing benchmark domain descriptions identified that MBP can represent and reason with domain knowledge that classical task refinement planners must resort to their precondition achievement function to achieve. Specifically, MBP can effectively represent and reason with expert knowledge for determining when actions should be included within a plan and derive dependency constraints from relationships between domain concepts. Classical planning offers the concepts of action preconditions and effects absent in MBP. These concepts facilitate the establishment and protection of causal links between actions

- **Integrated Architecture:** The integrated architecture exploits MBP constructs to represent and reason with domain expert knowledge. The actions and dependency constraints synthesised are compiled into task refinement schemas. A task refinement planner is then applied to combine the schemas into a complete and interaction free plan.

- **Evaluation:** The integrated architecture is evaluated from four perspectives. First, considering the complementary strengths that motivated the architecture identifies the facet(s) that address each issue. Second, a military evacuation scenario (Pacifica) is encoded within the formalism and a plan synthesised to test the generality of the architecture. Third, comments from automated planning experts obtained from publication of the architecture are examined and answered. Forth, construction industry experts' comments on the industrial potential of the architecture are discussed.

- **Conclusion and Further Work:** The integrated architecture successfully combines the expressive MBP representation and reasoning with the causal link establishment and maintenance of classical planning. Further work motivated by the integrated architecture lies in a number of directions: the integration of the domain knowledge from MBP within existing classical explanation techniques, further study of human inference and task knowledge, and tool support for building MBP domain models.

Table of Contents

Table of Figures

Dedication

To Ivy, Laurence, Betty, and Elizabeth.

Acknowledgements

Dr Graham Winstanley, The University of Brighton. For the initial direction and constant academic debate, guidance, and motivation without which this thesis would not have been written. I am in debt to you.

Dr Franco Civello, The University of Brighton. For the critical reviewing of this thesis and expert advice on the facets of object-oriented modelling.

Mr A. Rollings, Mr F. Parker, and Mr J. Simons, The Llewellyn Group of Companies. For supporting the knowledge acquisition and evaluation phases of this thesis.

Dr Brian Drabble, CIRL, University of Oregon, USA, formerly of AIAI, University of Edinburgh, UK. For discussions on the mechanics of the O-Plan planning system.

Engineering and Physical Sciences Research Council, Swindon, UK. For funding this research project (award ref. 94314463).

The following members of the planning community have either asked or answered questions which have had an input into the direction of this thesis:

Ruth Aylett, Jim Blythe, Jeff Dalton, Kutluhan Erol, Lois Prior, Earl Sacerdoti, Sam Steel, Austin Tate, and the anonymous reviews of the following conferences: AIPS-96, UK Planning and Scheduling SIG 96, ES-96, AAAI-97.

The following people have contributed their time, energy, support, comments or friendship in an important and appreciated way:

Richard Bosworth, M Cherif Bouzeghoub, Jonathon Dayao, Jon Durrant, Sarah Gordon, Richard Griffiths, Liz Guy, Victor González Laria, Jane Hudson, Martin Hunter, Araceli Hurtado, Dave King, Richard Mitchell, Jorge Núñez Suárez, Ian Oliver, Lyn Pemberton, Dan Simpson.

Updated to reflect examiners' comments November 11th 1997.

1. Introduction

If you ever get close to human behaviour be
ready to get confused....
Björk (1965 -)

1.1. Background to the problem

"Artificial Intelligence (AI) is a field aiming to achieve functionality in computers which, when exhibited by humans, is described as having indicated intelligence" (Brooks 1991). Planning is such a functionality and has therefore been studied by AI researchers since the field's inception in the nineteen fifties.

Informally, planning requires a functionality that can analyse an agent's environment and then develop a strategy that modifies that environment in accordance with the agent's goal(s) (Ginsberg 1993). This strategy is typically in the form of a set of actions combined with constraints on the order of their execution.

AI planning research has been based upon the hypothesis that planning applications, although diverse, share many common facets. It is therefore possible to build general-purpose planning systems that can process the specification of an application domain and then generate solutions to planning problems in that domain. This work is known as *domain-independent* planning.

Planning is an inherently complex problem. In order to provide an addressable subset of issues, the central focus of domain-independent research has made a number of simplifying assumptions known as the *classical assumptions*. Automated planning systems developed under these assumptions are termed *classical planners*. Whilst the development of classical planners has formed the core of automated planning research, the technology has not yet reached its industrial potential.

Model-based planning is characterised by the generation of plans from a central domain model of an organisation's products or services. The technology has developed independently from classical planning through an effort concerned with providing implemented applications for industrial clients. Whilst model-based planning has achieved successful commercial application, the technology's independence from classical planning has prohibited a cross fertilisation of ideas between the two approaches. This thesis examines both technologies, and develops an integrated architecture that exploits their relative strengths.

1.2. Aims and objectives

The aim of this thesis is:

- to develop the industrial aptitude of automated planning by combining classical and model-based technologies into a unified architecture which exploits each technology's strengths.

To achieve this aim, this thesis addresses the following objectives:

1. Identification of the complementary strengths and limitations of classical and model-based technologies.
2. Design and implementation of an integrated planning architecture which exploits the complementary strengths.
3. Evaluation of the resultant architecture against the rationale for its construction.

1.3. Summary of achievements (contribution to knowledge)

In terms of approach, the architecture developed in this thesis demonstrates the potential for collaboration between planning in the space of a static object-oriented domain model and planning in the space of an evolving partially-ordered state-based domain model. The object space exploits an expressive formalism for representing and reasoning with domain experts' knowledge. The state space facilitates the establishment of action conditions with action effects and the maintenance of these relationships by detecting and resolving interactions.

In terms of implementation, the integrated architecture details how task refinement schemata may be compiled from the object-based domain-modelling scheme of model-based planning.

The integrated architecture is summarised in Figure 1-1 below. The domain model provides object-oriented constructs for representing domain-specific knowledge. The model-based planner applies knowledge encoded in these constructs to determine actions and ordering constraints. The model-based/ HTN interface compiles the resultant structures into HTN schemata. The HTN-engine then assembles the schemata to create a complete plan. HTN critics are invoked to ensure the establishment and maintenance of conditions and effects over the plan.

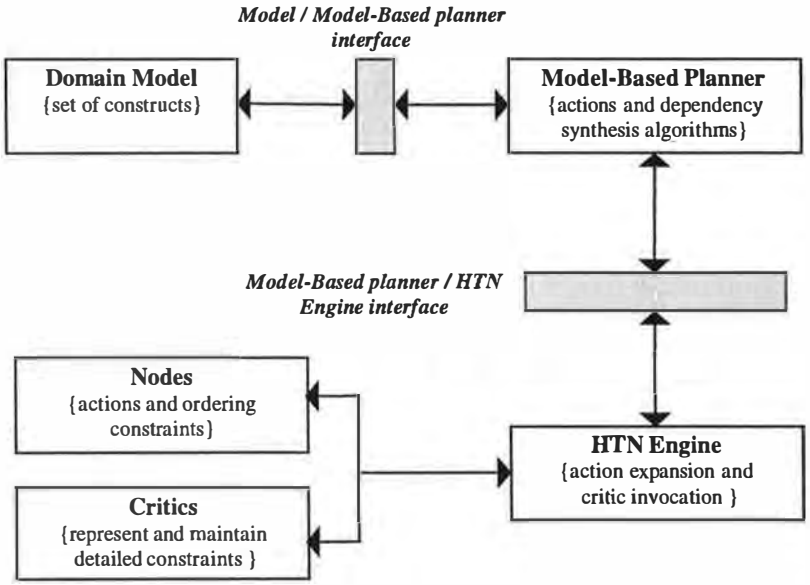


Figure 1-1 Overview of the integrated architecture

1.4. Approach and contents outline

Chapter 2. overviews classical and model-based planning technologies, emphasising the devices supported for specifying application domain specific knowledge. The chapter draws conclusions on the possibility of transferring ideas between the technologies, and the areas in which this transfer is likely.

Chapter 3. defines the need for a planning workbench to support the aims of this thesis, and details its implementation. Two appendices support the description with details of the algorithms implemented (Appendix A), and the test specification used to verify the workbench's correctness (Appendix B).

Chapter 4. builds upon the possible transfer of ideas proposed in Chapter 2 by identifying specific limitations with the technologies' representational devices. The limitations are identified through experiments within application domains detailed in the automated planning literature. This experimentation is supported by the research workbench described in Chapter 3.

Chapter 5. describes the elicitation of planning knowledge used by human experts within the construction industry.

Chapter 6. verifies and extends the set of limitations with classical and model-based planning technologies identified in Chapter 4 through a set of encodings of the construction domain elicited in Chapter 5. From the issues identified, a precise rationale for integrating classical and model-based planners is developed.

Chapter 7. describes an integrated architecture which exploits the relative capabilities of classical and model-based planning.

Chapter 8. evaluates the integrated architecture against the rationale for its development identified in Chapter 4 and Chapter 6, and assesses its commercial utility.

Chapter 9. summarises the research presented within this thesis, draws conclusions as to its success, and defines areas for further investigation.

1.5. Prerequisite knowledge

The reader is assumed to have the basic understanding of Artificial Intelligence that would be obtained by reading the introductory chapters of an Artificial Intelligence textbook. Examples of such texts include (Luger & Stubblefield 1993, Ginsberg 1993, Pratt 1994)

The reader is assumed to have the basic understanding of object-oriented design that would be obtained by reading the introductory chapter of an object orientation textbook such as (Wirfs-Brock, Wilkerson, & Wiener 1990). The commercial reader may wish to consider (Rumbaugh et. al. 1991) or (Booch 1991). The more formal reader may wish to consider (Cook & Daniels 1994)

Whilst automated planning theory is described from first principles, the reader would benefit from reading this thesis in combination with the excellent *Readings in Planning Volume* (Allen, Hendler, & Tate 1990). The volume contains many of the papers that define the foundations of planning theory referenced in this thesis.

1.6. Published work

The following publications resulted directly from the research presented in this thesis: (Jarvis & Winstanley 1996a, Jarvis & Winstanley 1996b).

This section is included in accordance with The University of Brighton's research degree regulations.

BLANK

2. AI planning technologies

Each generation imagines itself to be more
intelligent than the one that went before it, and wiser
than the one that comes after it.
George Orwell (1903-1950)

2.1 Introduction

This chapter overviews classical and model-based planning technologies, emphasising the devices supported for specifying application domain specific knowledge. The chapter draws conclusions on the possibility of transferring ideas between the technologies, and the areas in which this transfer is likely.

Classical planning systems are described in two stages. Section 2.3 covers the foundations of classical planning theory. Section 2.4 describes work that extends the foundation theory to provide techniques that are closer to the requirements of industrial planning applications. The overview concludes by bringing together views on the applicability of the two predominant classical planning techniques to industrial problems.

The overview begins with the definition of the problem used by early classical planning researchers. This definition is expanded in section 2.4.1 to provide a framework closer to the requirements of industrial applications. With classical and model-based technologies defined, the rationale for considering the transfer of ideas between the technologies is presented.

2.2 Initial definition of the planning problem

Figure 2-1 presents Kambhampati's definition of planning in terms of a dynamical system (Kambhampati 1996). The figure is useful for determining the facets of the planning problem that a domain independent planning system must consider.

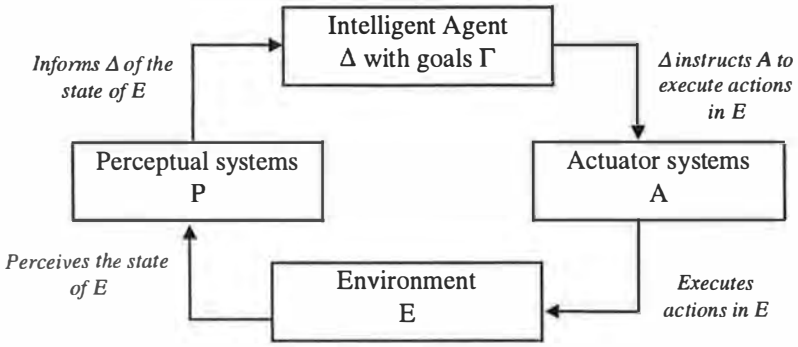


Figure 2-1, Planning in terms of a dynamical system

The agent Δ wishes to control the environment E in accordance with its own goal(s) Γ. The agent may perceive the state of E through its perceptual systems P, and it may effect changes on E through its actuator systems A. A plan is therefore a set of actions which when executed by A, modifies E in accordance with Γ.

To solve the planning problem, a domain-independent planning system must provide methods for addressing the following issues:

1. Representation of an agent's goals Γ.
2. Representation of the state of the environment E.
3. Representation of effects on E of the actions executable by A.
4. Provision of an algorithm to identify the subset of actions available to A which will manipulate E to achieve Γ.

Issues 1, 2, and 3 provide the requirement for a domain independent planning system's representational devices. Issue 4 specifies the requirement of a domain independent planning algorithm.

2.3 Foundations of classical planning

Within classical planning, the more expressive a language for representing the state of the world, the goal(s) of an agent, and the actions executable by an agent's actuators, the harder the task of writing a planning system to work with that language (Weld 1994). Consider the issue of representing actions with effects that vary depending upon the situation in which they are applied. An example action of this type is *release-car-foot-brake.* This action will produce the effect that a car is free to move if that car's hand brake is not engaged. If the car's hand brake is engaged, the action will have the effect of the car being secured only by it's hand brake. Compared to the case where an action's effects do not depend upon an action's context, with conditional effects the planning system must additionally determine the situation of the action's application and identify effects that are then applicable.

This is a common issue within artificial intelligence, and is known as the balance of epistemological and heuristic adequacy (balancing the ability to represent and reason about a problem with efficiency) (McCarthy & Hayes 1969). Classical planning problems are defined as problems which conform to a number of simplifying assumptions (Wilkins 1988, p 3); the aim being to provide problems which require a restricted epistemology that is heuristically adequate. Classical planners are domain independent systems, usually implemented on computer hardware, which address the set of classical planning problems (Wilkins 1988, p8).

Classical planning's limiting assumptions are as follows (based upon (Weld 1994)):

- **Atomic Time:** Execution of an action by an agent's actuators is indivisible and uninterruptible. Hence, actions maybe modelled as atomic transformation functions, changing the world state instantaneously. An effect of this assumption is that actions cannot be executed simultaneously, i.e. they must be supplied to the actuators one at a time.

- **Deterministic Effects:** The effect of an agent's actuators executing any action is a deterministic function of the action and the state of the world when the action is executed. This assumption prohibits the specification of probabilistic actions, such as tossing a coin.

- **Omniscience:** The agent has complete knowledge of the world and the effects on world state of the actions executable by an agent's actuators. The assumption implies that the agent's perceptual systems can accurately perceive all facts within the world, and that the agent possesses complete knowledge of the effects of its actuators upon that world.

- **Sole cause of change:** The world state changes only through the execution of actions by an agent's actuators or through predefined events (e.g. a bank opening at 09:00). Hence, no other agents unpredictably affect the world state and the world state cannot change on its own.

The classical assumptions collectively define a world that changes instantly, predictably, deterministically, and within the complete control of the agent performing the planning function. The 'real world' does not meet these constraints, but it is possible to formulate complex experimental worlds that do conform to them, and then to develop powerful planners to address these problems.

The following subsections review the main classical planning technologies: precondition achievement planning, and task refinement planning. Precondition achievement planning is further decomposed into plan space and state space planning. Each technology is briefly defined, before the methods available for specifying application domain-specific knowledge are examined. This section concludes by comparing and contrasting the representational devices identified, and within the context of current thinking, identifying the technique which best meets requirements of industrial planning problems.

2.3.1 Precondition achievement planning

The first and arguably the most influential planner, STRIPS (the Stanford Research Institute Problem Solver), was developed by Fikes and Nilsson in the late nineteen sixties (Fikes and Nilsson 1971). STRIPS casts the planning problem as a search through the space of possible world states, and is therefore classed as a state space planner. The overview of precondition achievement planners will commence with this class of planner, before moving to the more sophisticated plan space class.

2.3.1.1 State space precondition achievement planning

In state space planning, the initial world state (the way the environment is now) and the goal world state (the way the agent wishes the environment to be) of a problem are represented by logical sentences, and actions are described as state manipulation functions. From this formalisation, planning becomes the search of possible state transformations that start from the initial world state, and terminate when a path is found which reaches the goal world state.

There are two components to a STRIPS style planner; the state based planning algorithm and the STRIPS representation of actions. The latter providing the representational devices for making application domain specific knowledge available to the planning algorithm.

Fikes and Nilsson place the following requirements upon an action representation language:

> *... we must state the preconditions under which it [the action] is applicable and the effects on a world model schema.*
>
> *(Fikes and Nilsson 1971)*

If we define the world as being in a finite[1] state at any given moment in time, we can specify the preconditions of an operator as conditions which must hold in the world state, and effects as the way in which this state changes as a result of an operator's application. STRIPS achieves this function through operator specifications consisting of three fields, the first field specifying an action's preconditions and the second and third specifying an action's effects. An example STRIPS operator is depicted in Figure 2-2.

[1] I.e. the number of facts holding in a world state is not infinite.

```
Operator Push (?Object, ?From, ?To)
    Preconditions:  At-Robot (?From)
                    At (?Object, ?From)
    Effects:    Add:    At (?Object, ?To)
                        At-Robot (?To)
                Del:    At-Robot(?From)
                        At (?Object, ?From)
```

Figure 2-2, Example STRIPS operator, from (Fikes and Nilsson 1971)

The push operator describes an action, executed by a robot, which moves an object (*?Object*) from a location (*?From*) to a second location (*?To*). The preconditions specify that before the action can be executed, the robot and the object to be moved must be at the start location (*?From*). The operator's effects are sub divided into two fields: *Add* and *Del*, abbreviations for *Add List* and *Delete List* respectively. The *Add List* statements are added to the world model as a result of the action's execution. Hence, execution will result in the object (*?Object*) being at the goal location (*?To*) and the robot, having carried the object, also being at the goal location. The *Delete List* specifies statements that should be removed from the world as a result of the operator's execution. In the robot example, the object is no longer at its start location (*?From*) and, as the robot has moved with the object, the robot is also no longer at the start location.

The STRIPS representation provides a declarative mechanism for specifying an action for use in a planning system. This representation is seminal in planning research, and has underpinned much work since.

The second component to a STRIPS style planner is the planning algorithm. An example state based algorithm, published by Weld (Weld 1994), is specified in Figure 2-3 below. The algorithm is invoked with the parameters *plan-ss (Initial-State, Goal-State, Set-Of-Actions,[ ])*, where *Set-Of-Actions* is the set of actions available to the planning agent for execution by its actuators, and *[ ]* is an empty list.

```
Plan-ss (world-state, goal-list, set-of-actions, plan)
 1.  If world-state satisfies each conjunct in goal-state
 2.  then return plan
 3.  else let Act = choose from set-of-actions an action whose
precondition is satisfied by world-state.
 4.  if no such choice was possible then
 5.  return failure
 6.  else let S = the result of simulating the execution of Act
     in the world state and return plan (S, goal-state, set-of-actions, concatenate
     (plan, Act)
```

Figure 2-3, Simple state space planning algorithm

Step 1 defines the planner's success criteria. If all the statements in the goal-list (the goal state on first invocation) hold in the world state, then planning is complete and step 2 returns the plan. Thus, if on the first invocation all the conjuncts in goal state hold in the world state, the plan returned will consist of an empty list because no actions are required as no change to the world state is needed - the goal state holds in the initial state. Step 3 selects an action from the set of actions available, based on the action's preconditions holding in the current world state[2]. If more than one action is applicable, we can assume the planner records the choice point to permit backtracking. If no such choice was possible (step 5), the algorithm has reached a dead end in its search space, and should return to the previous back track point. Step 6 simulates the execution of the action selected, producing a new world state. The new world state consists of the previous world state in addition to the facts in the selected actions *add list*, differenced with the facts in the actions *delete list*.

The plan-ss algorithm will consider all the possible actions that may be executed from a state. It will therefore find a plan if one exists and is hence considered complete. Completeness is at the expense of time, as no predictions may be made on how long it will take to find a plan for a given problem (Weld 1994).

It is important to note that whilst the concept of state space planning is generic, a variety of methods have been developed to efficiently navigate the search space. The algorithm plan-ss is known as progression, as it synthesises a plan starting from the initial state and works towards the goal state. An alternative, regression (Waldinger 1977), plans by searching from the goal state backwards towards the initial state. STRIPS itself employed the search strategy of *means-end-analysis*, a technique originating in the theorem prover GPS (Newell & Simon 1961). These techniques direct the search space considered by the planner, and have proved more effective in a number of laboratory domains than progression.

[2]It is this behaviour which leads to the term precondition achievement planning. Both state space and plan space planners aim to ensure the preconditions of operators are satisfied.

2.3.1.1.1 Summary of the representational devices supported by state space planners

State space precondition achievement planners utilise a state-based model for specifying actions, agent's goals, and describing the world. Each category is summarised in the table below.

Agent's Goals	Specified as a set of logical statements that the planner must work to achieve. The agent's goals are collectively known as the goal world state.
World State	Represented as a set of logical sentences, which together specify the world at some moment in time. The state of the world at the start of planning is defined as the initial world state.
Actions	Defined as state manipulation functions. Actions have preconditions and effects. Preconditions specify the statements that must be true in the world state before an action can be executed. Effects specify the changes in the world state which result from an action's execution. The representation is known as the STRIPS representation.

In addition to the declarative knowledge representation techniques identified above, the domain writer may modify the domain independent planning algorithm to optimise it for the application domain under consideration. It is common practice for a domain writer to examine the execution of a planner on a test problem set, analyse the resultant search space, and then modify the algorithm and the action representation (Drummond 1994). The options available will be discussed in depth in the next section, 2.3.1.2

2.3.1.2 Plan space precondition achievement planning

The NOAH planning system (Sacerdoti 1975b) introduced several concepts utilised by modern planners in both the precondition achievement category and the task refinement category. One change initiated by NOAH was the reformulation of planning from the search through world states, to the search through plan space. In the plan space, nodes represent a partially specified plan, and the edges denote plan refinement operations such as the addition of an action to a plan. This space facilitated the development of more powerful planning algorithms. For example, the space permits actions to be added to a plan at any point, in contrast to the single insertion points offered in state space planning.

A second innovation of NOAH was partial-order planning, which lead to the least commitment paradigm[3] prevalent in state-of-the-art systems today. The innovation notes that a plan should only have ordering constraints added when there is a clear justification for them. In the total order approach found in state-space planning, ordering constraints are intertwined with the order in which actions are added to a plan. The Sussman Anomaly (Sussman 1974) provides a motivating problem that demonstrates the problems with the total order approach. The planner is provided with two goals *on (block1, block2)* and *on (block2 block3)*. If the planner attempts to realise the first goal before the second, it will find it can no longer move *block 2* onto *block 3*, as *block 1* is on top of *block 2*, preventing it from moving. The planner must then undo its first goal in order to achieve the second, before again achieving the first goal. By detecting that moving *block 1* onto *block 2* would delete the preconditions of the *move block 2 onto block 3* action, an appropriate ordering constraint may be added; therefore, removing the redundant action.

To achieve partial order planning we must refine our definition of a plan and a planning algorithm. Partial order planning considers a plan as a three tuple <A, O, L>, where A is the set of actions in a plan, O is a set of constraints over the execution order of A, and L forms a set of causal links (Weld 1994).

Consider the specification of A = {$Action_1$, $Action_2$, $Actions_3$} and O specified to be ($Action_1 < Action_2$, $Action_1 < Action_3$). This representation constrains $Action_1$ to occur before $Action_2$, and $Action_1$ to occur before $Action_3$. These constraints are said to be consistent, as at least one valid ordering exists. The semantics of the constraints are that $Action_2$ and $Action_3$ may be executed in any order relative to each other.

[3] The least commitment paradigm was also motivated by the MOLGEN planning system (Stefik 1981). In the context of biological experiment design, MOLGEN demonstrated the effectiveness of gradually constraining a variable's instantiation as opposed to immediately committing to an instantiation.

Least commitment planning requires the capability to trace past decisions and the reasons behind these decisions; if a planner makes a ladder available it is for a reason, and removing the ladder before it is needed would invalidate a plan. Sacerdoti identified this issue and included the Table of Multiple Effects in his NOAH system. The table recorded an entry for each expression that was asserted or denied by more than one node within a plan. A conflict was recognised when an expression that is asserted at some node is denied at a node that is not the asserting node's goal. Information recorded about why decisions have been taken during planning is referred to as a plan's Teleology (Sussman 1974).

Tate later devised the Causal Link within his Nonlin system (Tate 1977) to address a limitation of the table of multiple effects. The table of multiple effects considers effects at the node level. Consequently, it maintains all the effects from a producing node to its goal node. Typically, only a subset of the effects should be maintained. Causal Links provide a teleology structure with three fields: a pointer to a producer action A_p, a pointer to a consumer action A_c, and a proposition Q which is an effect of A_p and a precondition of A_c. Such links are represented mathematically as:

$$Ap \xrightarrow{Q} Ac$$

Causal links are used to detect when a new action introduced to a plan interferes with past decisions. Such an action is known as a threat. Mathematically A_t threatens $Ap \xrightarrow{Q} Ac$ when $O \cup (A_p < A_t < A_c)$ is consistent, and A_t has an effect $\neg Q$.

The concept of threat detection and removal was first implemented by Sacerdoti and refined by Tate within the Question and Answering component of Nonlin (Tate 1977). The famous Modal Truth Criterion (Chapman 1987) formalised the notation of threats and their removal through the construction of the formal TWEAK planner. The concept of plan critics and plan debugging originated with HACKER (Sussman 1974).

For ease of representation, plan space planning specifies planning problems as null plans. A null plan has two actions $A=(A_0, A_\infty)$, one ordering constraint $O = (A_0 < A_\infty)$, and no causal links, L= (). $Action_0$ is the start action of the plan and has no preconditions, its effects are used to specify the world's initial state. $Action_\infty$ has no effects, but its preconditions specify the goals of the planning problem.

The planning algorithm below (plan-ps) is a simple regressive algorithm, developed by Weld (Weld 1994). The algorithm searches null plans, making nondeterministic choices until all the conjuncts of every action's preconditions have been supported by a causal link, and all threatened links have been protected from possible interference. Plan-ps's first argument is a plan structure and the second is an agenda of goals that need to be supported by links. Initially each item on the agenda is a pair <Q $action_\infty$>, i.e. the preconditions of $action_\infty$, the goals of the plan.

Plan-ps (<A,O,L>,Agenda, Set Of Actions Available)

1. If Agenda is empty, return <A,O,L>
2. Let <Q, $Action_{needed}$> be a pair on the agenda
3. Let $Action_{added}$ = Chose an action that adds Q (either a newly instantiated action from Set Of Actions Available or an action already in A that can be ordered consistently prior to $Action_{needed}$. If no such choice is possible then return failure.
   Let L' = L $\cup$ {$Action_{added} \xrightarrow{Q} Action_{needed}$}, and let O' = O $\cup$ {$Action_{added} < Action_{needed}$}. If $Action_{added}$ is Newly instantiated, then A' = A $\cup$ {$Action_{added}$} and O' = O $\cup$ {$Action_0 < Action_{added} < Action_\infty$}. Otherwise, let A' = A.
4. Let agenda' = agenda - {<Q, $Action_{needed}$}
   If $Action_{added}$ is newly instantiated, then for each conjunct in, Q_i, of its precondition, add <Q_i, $Action_{added}$> to agenda'
5. For every action A_t that might threaten a causal link
   $A_p \xrightarrow{R} A_c$, add a consistent ordering constraint, either
   a) add $A_t < A_p$ to O'
   b) add $A_c < A_t$ to O'
   if neither constraint is consistent, then return failure.
6. Plan-ps(<A',O',L'>,agenda',Set Of Actions Available)

Figure 2-4, Simple plan space algorithm

Step1 specifies the planning algorithm's success criteria. If the agenda is empty, all the goals of the plan have been achieved. Step 2 selects an outstanding item for achievement. Step 3 selects a new action that adds the effect Q just selected from the agenda. Alternatively an action already existing in the plan, before A and asserting the effect Q, may be used to establish Q. The agenda item satisfied is removed from the agenda. If a new action has been instantiated, its preconditions are added to the agenda. Step 5, the causal link protection stage, moves actions that threaten other actions either before or after the range under protection. If either technique resolves the conflict, the plan is consistent and the algorithm is recursively called to solve the next agenda item.

The plan-ps algorithm contains several decision points: the selection of an entry from the agenda, the selection of an existing condition (if possible) to satisfy the agenda item selected against instantiating a new action, the choice of a new action from the set of possible actions, and the selection of a method to resolve a conflict. A more succinct decision is the choice of when to resolve the conflicts, the plan-ps algorithm hard codes the resolution of all conflicts created by a new action before considering the next agenda item.

Decision making strategies have been studied by a number of authors. To provide an insight into the decision-making strategies available, the work of Poet and Smith is summarised below.

Poet and Smith (Poet & Smith 1993)[4] identify several strategies for removing threats in partial order plan space planners. The *separable delay* technique notes that many of the threats that occur during planning are ephemeral (short-lived). As planning continues, new actions or variable bindings cause threats to be resolved without intervention from the planner. The *separable delay* strategy therefore waits until a threat has become definite i.e. all variable bindings have been completed. The *delay unforced threats* strategy waits until only one threat resolution option remains. Thus, if a threat could be resolved in plan-ps by both promotion and demotion, the planner would wait until only one of these methods would achieve a valid plan i.e. not introducing a new threat itself. *Delay resolvable threats* ignores a threat until it becomes impossible to resolve, and discards the partial plan. *Delay threats to the end* waits until planning is complete before attempting to resolve threats.

Joslin and Pollack (Joslin & Pollack 1994)[5] modify Poet and Smith's delay unforced threat strategy to one named *least cost flaw repair*, and apply it to both the selection of items from the agenda and selection of which flaws to repair. Flaws are ranked against the number of possible ways of resolving them. The strategy covers the contingency of all forced flaws being resolved and planning being complete except for a set of unforced flaws (flaws with a cost > 1). Least cost flaw repair processes each flaw (open condition or threat) in ascending cost order.

[4] A number of other authors have considered this issue (Yang & Chan 1994; Minton, Bresina & Drummond 1991; Barret & Weld 1994b)

[5] The authors later develop a technique criticising the class of decision postponements listed above, and introduce an active decision postponement technique (Joslin & Pollack 1995).

2.3.1.2.1 Summarising representational devices supported by plan space planners

Plan space planners enhance our understanding of the planning problem by refining the general-purpose planning algorithm. The recasting of planning as a search through the state of possible plans facilitates the introduction of non-linear planning, plan teleology, and threat handling. This reformulation of planning to the search through plan-space results in no enhancement to the declarative domain knowledge available to a domain writer; the STRIPS action representation remains unchanged.

As in state based planning, the domain writer has two methods available to specify domain specific knowledge: a declarative action description language, and algorithmic optimisation options. The declarative knowledge available to a plan space planner is defined in the table below.

Agent's Goals	Specified as a set of logical statements that the planner must work to achieve. The agents' goals are collectively known as the goal world state. In plan-space planning the goal world state is specified as preconditions on a final dummy action.
World State	Represented as a set of logical sentences that together specify the world at some moment in time. The state of the world at the start of planning is defined as the initial world state. In plan space planning the initial state is specified as effects of an initial dummy action.
Actions	Defined as state manipulation functions. Actions have preconditions and effects. Preconditions specify the statements that must be true in the world state before an action can be executed. Effects specify the changes in the world state which result from an action's execution. The representation is known as the STRIPS representation.

The algorithmic optimisation option permits the domain writer to make domain specific knowledge available to a planner through modifications to the planning algorithm. Plan space planners offer more modification options to a domain writer than state space planners. Specifically, state-space planners offer two decisions point for optimisation: the selection between the open conditions, and the selection between the applicable actions to achieve the condition selected. Plan space planning extends this set of decisions to include: the selection between the set of open conflicts in a plan, the selection between the set of methods to resolve a given conflict, and the decision to resolve conflicts or continue planning.

A number of decision-making strategies were described in the preceding text (separable delay, delay unforced threats, etc.). The method employed by domain writers for deciding between algorithmic optimisation options is one of experimentation. The domain writer encodes a domain in the declarative action representation language, then experiments with this representation on a number of example problems in the domain under consideration. "Bottlenecks" in the execution are identified, i.e. points at which the planner is exploring a large number of options whilst attempting to make a decision. The available decision making strategies are interchanged until a reduction in the identified bottleneck is achieved.

Drummond states that precondition achievement planning's applicability to real problems is hindered by the technique's search space (Drummond 1994). He notes that the technique's lack of effective search control prohibits the application of precondition achievement planners to real world domains. Drummond has identified a problem in the relationship between a domain and the allocation of algorithmic optimisation techniques. For example, there is no answer to the question of what features of a domain lead to the delay removable threats technique being more applicable than separable delay. Precondition achievement planners offer little advice to the domain writer; the decision must be based upon experimentation.

This argument will be considered in more depth when comparing precondition achievement and task refinement planning technologies in the next section.

2.3.2 Task refinement planning

Task refinement or Hierarchical Task Network Planners (HTN) (Tate 1977) were initially motivated by Tate's desire to combine AI planning techniques with Operations Research. Tate noted that the Operations Research discipline provided techniques for analysing plan networks, but offered no comment on their construction (Tate 1977). As a result of this heritage, there are many similarities between task refinement and precondition achievement planners, but also some significant differences. A comparison between the techniques is provided in section 2.3.3. This section aims to introduce the basic task refinement representation and planning algorithm.

Task refinement planning is the search for plans that accomplish task networks (Erol, Hendler & Nau 1993; Kambhampati 1994). The process of task refinement will be introduced through a simplified definition of task networks; therefore, for the moment, accept the following definitions.

- *primitive tasks* describe the actions available to the actuators of the agent for which we are planning.
- *non-primitive tasks* describe tasks that the planning process must accomplish. Non-primitive tasks are themselves composed of other non-primitive and / or primitive tasks, collectively defined as sub tasks - relative to the defining non-primitive task. A non-primitive task may place ordering constraints upon its sub tasks.
- *plan executability* is achieved when all the steps in a plan are mapped to primitive tasks.

Planning problems are supplied to a task refinement planner in the form of a non-primitive task network (a primitive task network is a completed plan, and would therefore require no action by the planner). Considering a concrete example, to generate a plan for decorating a house, the following task would be supplied to a task refinement planner (Figure 2-5):

```
task decorate;
 nodes  1    action {decorate};
end task decorate;.
```

Figure 2-5, Initial task network for the decorate problem

The initial task specification results in a plan with a single task: *decorate*. *Decorate* is non-primitive (defined by the prefix "action") and therefore requires further refinement. To refine the decorate task, the planner will search its domain library for a task network indexed as a refinement of *decorate*. Several task networks may share the same index, collectively referred to as methods. In the *decorate* example, one method may decorate a house with wall paper and a second with paint. Once the set of methods for achieving the decorate task has been identified, the planner selects one method from the set for use in the plan (the options available to guide this selection will be discussed latter). Figure 2-6 depicts two methods (*decorate 1* and *decorate 2*) which may be used to achieve the decorate task. *Decorate 1* is intended for a building with a basement, whilst *decorate 2* is intended for a building without a basement.

```
schema decorate-1;
 expands{decorate};
 nodes   1 action {fasten plaster},
         2 action {pour basement floor},
         3 action {lay finished flooring},
         4 action {finish carpentry},
         5 action {sand and varnish floors},
         6. action {paint};
 orderings: 2→3, 3→4, 4→5, 1→3, 6→5;
end schema decorate-1;

schema decorate-2;
 expands {decorate};
 nodes   1 action {fasten plaster},
         2 action {lay finished flooring},
         3 action {finish carpentry},
         4 action {sand and varnish floors},
         5 action {paint};
 orderings: 1→2, 2→3, 3→4, 1→2, 5→4;
end schema decorate-2;
```

Figure 2-6, Two methods for achieving the decorate task, based upon an example given in (Tate 1977)

Assuming the planner selects the *decorate-1* method, the initial decorate task will be refined to (i.e. removed and replaced by) the *decorate-1* method. Figure 2-7 depicts the initial plan and the results of the first refinement.

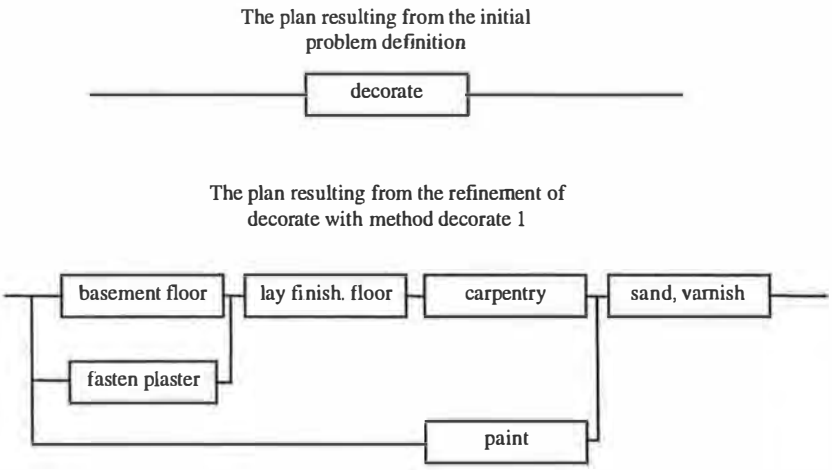


Figure 2-7, Initial refinement of the *decorate* task.

Each of the tasks resulting from the initial refinement of the decorate task are non-primitive, as each task has the prefix "action". Hence, the planner proceeds by searching for methods in its domain library that will achieve each of the new tasks (*fasten plaster, pour basement floor... paint*), and selecting appropriate refinements. If the *paint* task has one method applicable for its refinement (Figure 2-8), the task network in Figure 2-7 will be refined to the network depicted in Figure 2-9. Note how the ordering constraints imposed on paint are maintained by the task's refinement.

```
schema paint;
 expands {paint};
 nodes  1 primitive {paint walls},
        2 primitive {paint door frames},
        3 primitive {paint doors};
 orderings: 1→ 2;
end schema paint;
```

Figure 2-8, Refinement for the *paint* task

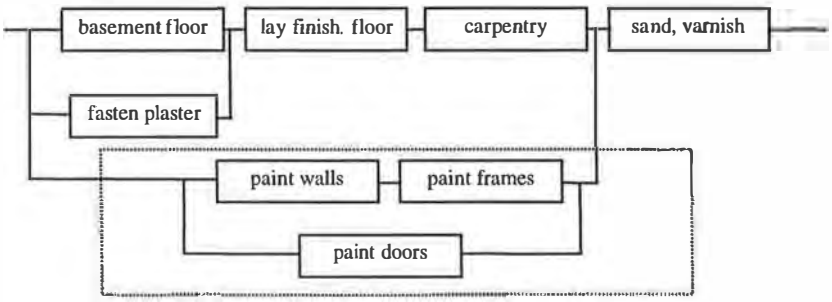


Figure 2-9, Plan resulting from the refinement of the *paint* task

The refinement process repeats until each task in the a plan may not be further refined, i.e. all tasks are primitive tasks. The process of task refinement is described more formally in the algorithm below (Figure 2-10) which originates in (Erol, Hendler & Nau 1993).

```
Plan-tn (Initial Task)
 1. If P contains only primitive tasks, then resolve the conflicts
    in P and return the result.
    If the conflicts cannot be resolved, return failure
 2. Choose a non-primitive task t in P
 3. Choose an expansion for t
 4. Replace t with the expansion
 5. Use critics to find the interactions among the tasks in P
    and suggest ways to handle them
 6. Apply one of the ways suggested in step 5
 7. Go to step 1.
```

Figure 2-10, Simple task refinement planning algorithm

On the algorithm's first invocation, step 1 considers the initial task. If all tasks in the plan are primitive, planning is considered complete. Step 2 selects a non-primitive task from the initial task specification; hence, it is possible to describe problems that initially consist of more than one task. Step 3 identifies methods which maybe used to refine the task selected, and selects one to use as a refinement. Step 4 performs the refinement by replacing the task selected for refinement with the selected method for accomplishing it. Step 5 and 6 critique the plan for inconsistencies in the constraints explained below. Step 7 recursively calls the algorithm. Any non-primitive tasks added during the previous invocation of the algorithm will be considered at step 2 for refinement.

As with precondition achievement planning algorithms, the task refinement algorithm in Figure 2-10 offers several decision points: should all tasks be refined before constraints are checked or, as in the algorithm above, should the process of task refinement and constraint maintenance be interleaved? If a set of tasks require refinement, which of these tasks should be considered first. Tsuneto et. al. identify the choice points in task refinement planning and analyse the options available for making these choices (Tsuneto et. al. 1996). The options correlate with Poet and Smith's work in precondition achievement planning (Poet & Smith 1993).

To date, this description has described non-primitive task networks as structures containing a set of other non-primitive and or primitive task networks and ordering constraints between these sub tasks. The domain writer may specify additionally knowledge in the following categories: conditions, effects, and variable binding constraints. Each category is described in turn below.

Conditions may be typed to permit the domain writer to define how each should be established and maintained. The variety and meaning of condition types vary from one specific planning system to another. For the purpose of this review, the types in the Nonlin system will be examined. Nonlin was the first task refinement planner[6]; therefore, the condition types available in today's state of the art systems may be traced to those provided in Nonlin.

Nonlin supports *achievable goals* and three condition types: *supervised*, *unsupervised*, and *use-when* (Tate 1976). Achievable goals permit Nonlin to include tasks in its network for to purpose of satisfying the condition the goal prefixes. This is the only construct that permits Nonlin to include tasks for satisfying a condition. The supervised condition type permits the specification of the only goal node from which a condition may be satisfied[7].

Unsupervised conditions may be established only by adding ordering constraints to a plan. A planner will not add actions to a plan for the purpose of establish conditions of this type. In Nonlin, the system waits until planning is complete before attempting to match each unsupervised condition with a contributor. If contributors cannot be found, Nonlin backtracks by trying different refinements for tasks.

Use-when conditions provide a mechanism for selecting between the set of methods available for achieving a task. For example, if a decorate schema is only applicable to a house with a basement, a use-when condition of "building has a basement = true" would be added. Once the set of methods for refining a task is identified, the members whose use-when conditions do not hold in the current world state are pruned. A task refinement planner will not attempt to satisfy a use-when condition, hence, the planner will not attempt to construct a basement for decorating a house.

[6]Nonlin draws upon concepts originating in Sacerdoti's NOAH system. However, Nonlin was the first planning system to implement task refinement planning as found in today's planning systems.

[7] The *achievable goal* concept has been replaced within the O-Plan system with the condition type *achieve*. In O-Plan *Supervised* conditions may be satisfied within a schema either by the deliberate inclusion of an effect or by the direct inclusion (at a more detailed level of decomposition) of an action known to achieve the necessary effect.

Effects are similar to those in precondition achievement planning. They describe how the world state changes as a result of an action's execution. As tasks may be organised into a hierarchy, effects may be organised into different levels of abstraction.

Variables may have co-designation constraints placed upon them.

With the additional constraints specified, a complete definition of primitive and non-primitive tasks may be presented.

- *primitive tasks* describe the actions available to the actuators of the agent for which we are planning. They may assert effects and conditions upon the world.
- *non-primitive tasks* describe tasks which the planning process must accomplish. Non-primitive tasks are themselves composed of other non-primitive and or primitive tasks, collectively defined as sub tasks - relative to the defining non-primitive task. A non-primitive task may assert the following constraints: orderings between its sub tasks, typed conditions, and variable binding constraints.

Task refinement planning is the only technique which, to date, has successfully been applied to "real world" planning problems (Kambhampati 1995). In comparison with precondition achievement planning, there is a lack of formal understanding of task refinement planning. Until recently, many authors have dismissed task refinement as an efficiency hack. However, Erol and Kambhampati have recently made progress towards developing a formal understanding of task refinement planning.

2.3.2.1 Summary of the representational devices supported by task refinement planners

Task refinement planning offers a number of constructs to the domain writer for declaring knowledge about actions in a domain. Each is summarised in the table below:

Agent's Goals	Planning problems are specified as one or more non-primitive tasks which the planner must refine. Ordering constraints between these tasks may be specified by the domain writer.
World State	Represented as a set of logical sentences, which together specify the world at some moment in time. The state of the world at the start of planning is defined as the initial world state.
Actions	A domain is partitioned into a set of tasks, with a number of methods for achieving each task. The tasks may be arranged into a hierarchy, with each level representing a different level of abstraction. Preconditions may be typed as achievable goals, supervised, unsupervised, or use-when. Condition types inform the planner how to satisfy and maintain conditions, therefore reducing the planner's search space. Effects of actions may be specified as in precondition achievement planning, as literals which change in the world state. The hierarchy of task specifications permit effects to be introduced at different levels of abstraction.

The domain writer may modify a task refinement planner's decision-making strategies in a number of ways. The options available are similar to those in precondition achievement planning. This point is expanded in section 2.3.3.2.

2.3.3 Summary of the views on which classical planning technology best meets the requirements of industrial planning problems

Sections 2.3.1 and 2.3.2 identified two devices supported by both precondition achievement and task refinement technologies for providing application domain specific knowledge to a general-purpose planning system. First, *declarative action representations* allow the domain writer to declare knowledge about the actions and the objects in a domain. Second, *algorithm optimisation* techniques permit the domain writer to adjust a general-purpose planning algorithm to optimise the algorithm's performance for a specific application domain.

Both methods are summarised below, and conclusions drawn as to which technology, precondition achievement or task refinement, best services the representational requirements of industrial planning problems.

2.3.3.1 Declarative action representations compared

Both state space and plan space approaches to precondition achievement planning support the declarative STRIPS action representation language. Within the STRIPS formalism, the world is modelled as being in a finite state at any moment in time. Actions are defined in terms of preconditions and effects, where preconditions define the conditions which must hold in the world state before an action can be executed, and effects specify the logical statements added to and deleted from the world's state as a result of the action's execution.

As in precondition achievement planning, task refinement planners use a state-based model as the underlying action description technology. The primary differences lie in the way action knowledge is organised and the type prefixes that may be added to conditions. Actions are organised into a hierarchy of tasks, where each task may define its sub tasks, ordering constraints between those sub tasks, conditions, effects, and variable binding constraints.

The differences between the declarative action representations of precondition achievement and task refinement planning lead to the question: "which representation is more expressive[8]?"

[8]In terms of features which are relevant to capturing domain specific knowledge in the context of solving the planning problem.

From task refinement planning's conception in the nineteen seventies to the late nineteen eighties, no formal analysis of the task decomposition was undertaken (Barrett & Weld 1994b; Kambhampati 1994; Erol & Hendler & Nau 1994b). Formal research focused on precondition achievement planning, neglecting task refinement (Chapman 1987; Pednault 1988; McAllester & Rosenblitt 1991). Recently (the early nineteen nineties), researchers have started formally analysing the implications of task refinement. Work in this area has been undertaken by (Yang 1990; Erol & Hendler & Nau 1994a, 1994b; Kambhampati 1994, 1995; Barrett & Weld 1994).

Erol et. al. (1994b) use Baader's definition of the expressivity of knowledge representation languages to compare precondition achievement and task refinement planning. Baader defines expressivity as follows. If a language L_1 can be expressed in a second language L_2, then for any set of sentences in L_1, there must be a corresponding set in L_2 (Baader 1990). Erol et. al. demonstrate through a formalisation of task refinement and precondition achievement planning that the precondition achievement representation can be expressed in a task refinement formalism (Erol, Hendler & Nau 1994b). The authors then demonstrate that the inverse is not possible, i.e. all sentences in a task refinement formalism cannot be represented in a precondition achievement formalism. Therefore, under Baader's definition of language expressivity, precondition achievement formalisms are less expressive than task refinement formalisms. Erol et. al. (1994b) conclude that precondition achievement planning is a special case of task refinement planning. This analysis does not comment on the relevance of task refinements greater expressivity in the context of capturing domain specific knowledge for a planning system.

Kambhampati (Kambhampati 1995) derives a formal framework that is similar to Erol et. al.; however, he uses the framework to examine informal claims made about task refinement's advantages by a number of researchers in the context of planning. Kambhampati's analysis is summarised below:

It is often claimed that task refinement planners allow the domain writer to effect more control over solutions than precondition achievement planning, as the domain writer may rule out certain classes of solution through task specification. For example, by specifying the task "build a house" rather than the goal "have a house", the planner does not explore the options of purchasing a house or moving an existing house to a new location. Kambhampati questions if the same functionality may be achieved in precondition achievement planning. Barret and Weld (Barret and Weld 1994a) offer such a comparison and conclude that whilst many of the problem specification advantages can be achieved in precondition achievement planners, task refinement planning is the more expressive.

Task refinement planners encode large plan fragments with pre-packaged causal structure, hence, the planner does not have to work to create these plan fragments. Kambhampati concludes that this advantage depends upon the level of interaction between customised plans. Hence task refinement planners rely on addressing a class of problem where the domain maybe structured into a relatively interaction free task specifications.

Goal specification is arguably richer in task refinement planning as it is possible to specify intermediate goals. For example, a round trip cannot be specified using goals of attainment as the goal state and initial state are the same. Task refinement planning permits the problem to be specified as two ordered tasks "travel to location1 < travel back to original location". Kambhampati notes that problems of this type may be specified in precondition achievement planning by inserting a dummy action. However, it is not possible in precondition achievement planning to enforce restrictions on different parts of the plan. For example, enforcing the constraint that the round trip should use the same mode of transportation on both legs. A solution in precondition achievement planning would require modification to the domain model, where task refinement systems allow a new high level operator to be added, leaving the remainder of the domain model unmodified. Kambhampati concludes that problem specification is simpler in task refinement planning.

Task refinement planning is the only planning technique to be applied successfully in real world problems. Kambhampati asks if this is necessarily so. Drummond (Drummond 1994) argues that this practical success is no accident, attributing the success to task refinements ability to make pertinent domain knowledge available to a planning system. This is an unresolved point of contention.

Figure 2-11, Kambhampati's analysis of task refinement planning's expressivity

Current formal views agree that task refinement planning offers more features to a domain writer than precondition achievement planning. Erol's mechanistic analysis of the techniques demonstrates that these features lead task refinement planning to be the more expressive technology. Kambhampati's reasoned examination of task refinement's additional features indicates that the approach has two types of benefit. First, some features of the industrial application domains may be represented more simply in task refinement planning. Second, task refinement planning can represent features of industrial application domains that cannot be realised in any other technique.

McDermott provides a summary of the relationships between task refinement and precondition achievement planning (below).

> *"... The truth is that [precondition] and [refinement] planners are not competing. The spaces searched by [refinement] planners are quite different to those searched by [precondition] ones. A [refinement] planner pastes together big canned plans, postponing decision about how those plans will interact. That approach makes no sense unless each of the plans is written in a robust way that will allow it to succeed when other things are happening. That gives the planner the freedom to ignore most interactions. In other words, the planner is not avoiding interactions by means other than search; instead, it is presupposing that plans have been written so that fatal interactions are improbable. This presupposition is false in the blocks world, where all the difficulties are due to intricate combinatorics in stringing together tiny pieces of plan." (McDermott 1991)*

McDermott's arguments are supported by Drabble and Tate's description of the target applications for their O-Plan task refinement planner (Drabble & Tate 1994). The authors provide a taxonomy of problems which ranges from the resource intensive scheduling problems to the interaction intensive puzzles, such as blocks world. O-Plan's target applications are defined as residing in the centre of this continuum. The authors claim that many industrial applications of planning technology lie in this area.

Combining the arguments above with the pragmatic observation that task refinement planners have been applied more successfully to industrial problems, one must conclude that the representational devices supported by task refinement planning are the most effective classical offering for addressing industrial problems.

2.3.3.2 Optimisation of planning algorithms compared

The table below demonstrates that the decision points within task refinement and precondition achievement planning are comparable.

Precondition choice	Task refinement choice point
Selecting an item from the set of items on the agenda for achievement	Selecting the next task from the set of non-primitive tasks in a network for refinement
Selecting an action from the set of actions available which achieves the current goal	Selecting a task from the set of methods available which refine that task
Should all conflicts be refined before the next goal is addressed, or should the planner wait until planning is complete	Should conflicts be addressed before the next task is refined, or should the planner wait until planning is complete
Of the possible ways to resolve a conflict, which should be implemented	Of the possible ways to resolve a conflict, which should be implemented.

Each of the decisions in the table above can be made using one of a number of strategies. Each strategy offers a different trade off between the time taken to compute the cost of each option available to the quick, but unsophisticated, random choice. Task refinement planning, however, provides a number of constructs in its declarative domain representation formalism for influencing each decision point. The mapping between decision point and construct is depicted in the table below.

Task refinement choice point	Declarative construct
Selecting a task network to establish an effect.	Effects may be typed to differentiate between the effects for which an operator should be included in a plan from the side effects of an operator.
Selecting a task from the set of methods available which refine that task	Filter conditions define the criteria which must hold before a method is applicable. The construct permits candidate methods for inclusion in a plan to be discounted.
Should conflicts be addressed before the next task is refined, or should the planner wait until planning is complete	Condition typing constrains the order in which conditions should be addressed.
Of the possible ways to resolve a conflict, which should be implemented.	Condition typing constrains the method employed to establish each condition.

In contrast to precondition achievement planning, task refinement planning provides a domain writer with devices for controlling the execution of the domain-independent planning algorithm's execution.

2.3.3.3 Conclusion

Precondition achievement is the most general classical planning technology. The precondition - effect representation places no assumptions upon the structure of the domains to which it is applicable, leaving the complexity of planning to the domain-independent planning algorithm. The technique's industrial aptitude is limited, however, by the large search space the planning algorithm must consider and the lack of domain independent heuristics for reducing that space.

Task refinement planning offers a formalism that permits the structure of a domain to be exploited, hence, addressing partially the prohibitive search of precondition achievement planning. Task refinement's additional features place certain assumptions upon the application domains to which it is applied. Specifically, that the application domain can be encoded as a set of task networks with limited interactions.

The assumption placed upon application domains by task refinement techniques has held in a number of industrial applications. Hence, combined with the inclusion of precondition achievement functionality in task refinement planning, if one wishes to consider an industrial planning domain, task refinement technology should be considered first.

2.4 Extending the classical framework[9]

The classical assumptions have provided a manageable subset of real world issues for planning researchers to address. However, the techniques developed within this framework are limited in their application as "real world" domains do not conform to the classical assumptions. In recent years, the ARPA[10] / Rome Laboratory Planning Initiative (ARPI) (Tate 1996b) has provided the largest single funding source for AI planning research, and has therefore effected a major influence on the direction in which the field has progressed. The initiative summarises its participants' perception of the status of planning research at its conception (in the late nineteen eighties) in the quotation below:

> *The AI planning community believes that it has many of the constituent theories in place, but what has yet to be demonstrated is what is important and what is not* [11]*. reported in (Fowler et. al. 1996).*

Erol, an ARPI participant, confirms this view with the following comment:

> *The current state of the art in planning research has not yet reached a level to accommodate the demands of the planning applications. Developing fast, reliable planning systems that work well in planning applications is still a great challenge for planning researchers.*
>
> (Erol 1995, pp 124)

The ARPI initiative has aimed to move planning technology closer to the demands of industrial problems by biasing its funding towards research with an industrial focus, hence, encouraging planning researchers to relax the classical assumptions and identify and develop what Fowler reports as the important theories.

To provide a framework for the presentation of state of the art planning research, the definition of the planning problem in Figure 2-1 is extended in Figure 2-12 to provide a more general definition. State of the art research is then briefly summarised against this framework, before conclusions are drawn as to how current work is developing the domain knowledge available to planning systems.

---

[9] This title is inspired by a chapter title in (Allen, Hendler & Tate, 1990).

[10] The Advanced Research Project Agency (ARPA) has recently been renamed the Defence Advanced Research Project Agency (DARPA).

[11] This quotation is cited in (Fowler et. al.1996), but is described as originating in an unpublished technical report. Hence, the citation is attributed to Fowler et. al.

2.4.1 General definition of the planning problem

To provide a framework for the analysis of integrated planning systems, the definition of a dynamical system in Figure 2-1 is extended in Figure 2-12 below.

The intelligent agent Δ has a set of goals Γ that indicate desires on the state of the environment E. The agent possess two subsystems for reasoning about the actions its actuator systems may take: a planning system and a reactive system. The planning system provides the long term strategic reasoning of the agent, whilst the reactive system supports immediate behaviour (in a robot this would equate to swerving to avoid an obstacle[12]).

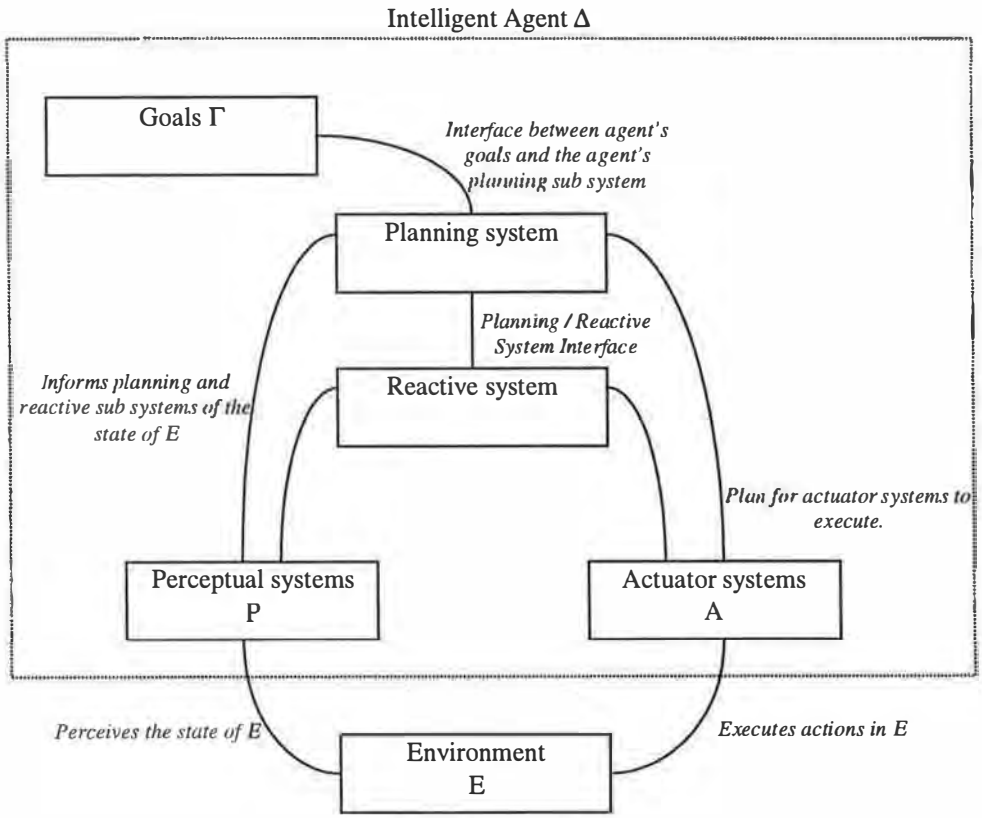


Figure 2-12, Generic planning architecture

The following summary of state of the art planning research will draw upon this definition of planning to set the work described into the context of the overall-planning problem.

[12] The need for a reactive and a planning component is a contentious issue. Brooks (Brooks 1985; 1991a; 1991b) argues intelligent robots require only reactive behaviour. Whilst Ginsberg (Ginsberg 1989) argues that it is impossible to encode every situation an agent may find itself in, therefore planning functionality is required.

2.4.2 Overview of current research directions

2.4.2.1 Relationships between planning, acting, and reacting

If the assumption of complete world knowledge is relaxed, during plan execution an agent may discover its initial understanding of the world was incorrect or incomplete (Beetz & McDermott 1996). This scenario poses the question of when to stop executing a plan as a result of discovering new knowledge, metaphorically taking a step back to consider the option of replanning as opposed to continuing the execution of the existing plan. A related concern is when to plan and when to react.

The relationships between planning, acting, reacting, and replanning have formed a major thread in AI planning research. The majority of such work has been driven by robotics applications where there is a great emphasis on the need to react and re-plan (Wilkins & Myers 1995). The robot work is typified by McDermott's reactive planning language (RPL) (McDermott 1992; Beetz & McDermott 1996). RPL originally provided a language for specifying how a robot should respond to sensory input in order to accomplish its tasks. Beetz and McDermott's recent work extends RPL to allow a robot to make decisions about when to replan, how to continue a task when waiting for a plan, and how to integrate new plans into the plan a robot is currently executing.

Task refinement researchers identified the need to consider the relationship between a planner and its execution whilst addressing a military logistics domain (Wilkins & Desimone 1994). The work has resulted in the SIPE-2 (Wilkins 1988) task refinement planner being integrated with the Procedural Reasoning System (PRS) (Georgeff & Ingrand 1989) reactive system, creating the CYPRESS system (Wilkins & Myers 1995). Similar research has been undertaken on the O-Plan project in the context of a military evacuation domain (Tate 1993a). The essence of this research is defining the communication between a planner and an execution system. With particular emphasis on how a execution system may integrate a new plan fragment into its existing plan, and the decision of when a plan must be terminated and replanning initiated. Other research in this area is briefly summarised below:

- Classical planning implicitly assumes a planner output provides instructions for a single agent. Work by (Coddington & Aylett 1996) addresses the issue of planning to co-ordinating many robots. Related to this issue, is planning for environments in which an agent is competing with other agents (Fuchs 1996).

- Blythe is working in domains where the environment changes independently of the agent the planner controls (Blythe 1996). Blythe is working with the management of oil spills, where the weather may change, moving the spill into a different direction.

2.4.2.2 Developing the expressive power of planning languages

Planners require languages that can express knowledge about actions. Work in this area has proceeded in task refinement and precondition achievement planning. Advancements within each technology are summarised below.

Precondition achievement planning

In the early nineteen nineties, precondition achievement planning could be divided into two camps. In the first group, formally complete partial-order planners could reason with a restricted form of the STRIPS representation. E.g. TWEAK (Chapman 1987), SNLP (McAllester & Rosenblitt 1991). In the second group, formally complete planners could reason with relatively expressive formalisms, but could work only with totally-ordered plans. E.g. Pedestal (McDermott 1991).

Since partial order planning is preferable to total order approaches (Minton et al 1991)[13], planning researchers have aimed to produce a formally complete, partial-order planner that would support an expressive action representation.

Pednault's Action Description Language (ADL) is an expressive planning language, designed to integrate the advantages of the STRIPS representation and the situation calculus (Pednault 1989). Pednault effectively reformulated the situation calculus into action schemas akin to those in the STRIPS representation. This reformulation resulted in a formalism more expressive than STRIPS yet computationally less demanding than full first order logic. McDermott's Pedestal planner was the first implementation of this language (McDermott 1991). Pedestal used a total order plan representation, and McDermott argued the total order approach was the only way to realise ADL. UCPOP (Penberthy & Weld 1992) provides the first partial-order implementation of a significant subset of ADL. More specifically, UCPOP can represent actions with conditional effects, universally quantified preconditions and effects, and universally quantified goals. UCPOP has been proved both formally sound and complete.

[13] (Minton, Bresina & Drummond 1994) demonstrate that the search space of a partial-order planner is never larger than a total order planner. In some cases, it is exponentially smaller. Hence partial order planning is generally more efficient than total order planning.

UCPOP has provided an expressive and sound base upon which researchers may develop planning technologies. The planner is currently in use in approximately one hundred institutions (Weld 1996). A common research paradigm is demonstrated by Weld & Etzioni (Weld & Etzioni 1994). The World Wide Web provides planners with a real world domain with which they may interact with sensors (e.g. gopher) and actuators (e.g. ftp). Weld & Etzioni noted the importance of safety in this domain. For example, a planner should be prevented from producing a plan which deletes the files on a disk in order to optimise the amount of free space. Weld & Etzioni add the concept of safety and tidiness to the action representation of UCPOP and successfully addressed these issues. This research paradigm may be summarised as, apply a planner to a new domain, identify requirements the planner cannot represent, modify the planner to represent the new requirements.

Other examples of work based on UCPOP are summarised below:

- **XII** Integrates UCPOP with an execution environment (Golden, Etzioni & Weld 1994)
- **BURDIAN** modifies UCPOP to represent actions with probabilistic effects. (Kushmerick, Hanks & Weld 1995)
- **PYRRHUS** balances the cost of a plan against the degree of goal satisfaction. (Haddaway & Hanks 1992)

It is important to note planners other than UCPOP are being utilised within the "apply and develop" paradigm. Most notably PRODIGY (Minton et al 1989). There is evidence to suggest more projects are moving towards UCPOP (e.g. (Knoblock 1996)).

Task refinement planning

The task refinement paradigm is being developed through two practically oriented projects: SIPE-2 (Wilkins 1988) and O-Plan[14] (Currie & Tate 1991).

O-Plan aims to provide a generic planning architecture which permits the "plug and play" of individual components (Tate 1993b). The rationale is, to allow the system's optimisation for specific domains. Much of O-Plan's developments are covered in other sections of this review (2.4.2.1, 2.4.2.3, 2.4.2.4). The expressive power of O-Plan's domain representation language (the task formalism) has been enhanced in the following ways:

- **Resource Types**. Vere demonstrated that Nonlin may be extended to allow specification of goals with relation to time and events (Vere 1983). Work on the O-Plan project has developed a rich model of resources to aid search control.

- **Refinement of Nonlin's task formalism.** The O-Plan project has refined the original task formalism implemented in Nonlin. Notably the semantics of condition types have been clarified (Tate & Drabble & Dalton 1994) and a full specification of the task formalism published (TFMANUAL).

[14] O-Plan is in its second incarnation and until recently was referred to as O-Plan2. The team has reverted to the O-Plan name to prevent confusion with implementation versions. (Private correspondence with Brian Drabble, formerly AIAI Edinburgh, UK).

2.4.2.3 Relationship between an agent and its planning subsystem

The interface between an agent's goals and its planning component is a significant issue in applications where a human user must interact with a software system. This scenario has led to the development of the mixed initiative planning paradigm.

O-Plan implements mixed initiative planning with user and planner co-operating to solve a problem with the planner asking questions of the user and the user placing tasks onto the planner's agenda (Tate 1994). The O-Plan scenario has been applied to military logistics planning (Tate, Drabble & Kirby 1994b)

The TRAINS-95 system concentrates on the communication media between planner and human (Ferguson, Allen & Miller 1996). TRAINS-95 uses speech recognition, graphical representations of domain concepts and natural language understanding to enter into a natural dialogue with the human user. Tate reports work that is combining the planning strengths of O-Plan with the user interaction features of the TRAINS project (Tate 1997).

2.4.2.4 Plan quality

As planners address real world domains, the resultant plans increase in complexity. It is difficult for humans to inspect such plans. Two complementary techniques have been developed to support this analysis.

Simulation systems allow a planner's execution environment to be simulated. MESS (Multiple Event Stream simulator) (Anderson & Cohen 1996) provides a domain independent simulation environment allowing streams of events to be supplied to a planner and the resultant plans analysed.

A number of valid plans may be produced to solve a single problem. Typically such plans vary in the number and type of resources used and the execution time of the plan. Each valid plan for a problem is referred to as a possible course of action (COA). Swartout and Gil provide a course of action evaluator that ranks the different courses of action according to domain specific criteria (Swartout & Gil 1996). The COA evaluator is combined with the EXPECT Knowledge acquisition tool to allow us to quickly build plan ranking criteria.

In (Drabble, Gil & Tate 1995) the EXPECT system is applied to a military domain, where plans are evaluated against the number of sea ports, air ports, flights per hour, and other domain specific criteria. This type of analysis integrates with the mixed initiative planning paradigm. In the military scenario, the user may select a course of action and ask the planning system to modify it under certain criteria. For example, reduce the number of aircraft resources required.

2.4.2.5 Developing planner knowledge bases

The construction, debugging, verification, and maintenance of planning knowledge bases has until recently been neglected. Chien provides a set of tools to support these tasks (Chien 1996). Specifically, Chien's tools tackle the problems of incorrect plan generation and the failure to generate a plan.

Wang addresses the construction of knowledge bases through a *learn by doing paradigm* in the OBSERVER system (Wang 1994; 1996). OBSERVER learns operators from sample problem solutions.

2.5 Model-based planning

Model-based planning systems (Marshall et al. 1987, Winstanley et al. 1990, Winstanley & Hoshi 1993) were developed during the late nineteen eighties and early nineties in a collaboration between The University of Brighton, UK, Stanford University, USA, Rediffusion Simulation Ltd, UK, and Babcock Woodall-Duckham Ltd, UK. The work addressed the design to implementation process of large scale, high technology products.

Model-based planning systems form part of a larger configuration and planning system. The configuration subsystem questions a designer with the factors affecting component choice. An example of this dialogue from the PIPPA (Marshall 1988) system working in the flight simulator domain is depicted in Figure 2-13 below. The configuration system applied rules of the form depicted in Figure 2-14 to select the components for a specific product.

Is the simulator to be situated in an unusual or hostile environment?[yes, no, maybe]
user> yes
Is the site subject to temperature extremes?[yes, no]
user >yes
Enter the name of the cooling technique which is to be used [oil, gas]
user >oil
How far away from the cooling system will the simulator be in meters
user >35
Conclusion: the simulator will require a HTU cooling unit with a booster pump due to the distance between the cooling system and the simulator.

Figure 2-13, Example configuration dialogue between the PIPPA configuration sub system and a flight simulator designer (from (Marshall 1988, pp 33))

If the environmental conditions of the simulator site are temperature hostile
then the piping material of the plumbing is stainless_steel
If the environmental conditions of the simulator site are humid
then the piping material of the plumbing is rubber.

Figure 2-14, Sample configuration rule set from the PIPPA configuration sub system (From (Marshall 1988, pp34))

The model of a product resulting from this configuration process is depicted in Figure 2-15 below. The *mechanical_system* and *hydraulic_comp* ovals represent classes of component. Whilst the *acuator_1, upper_eye_end, lower_eye_end,* and *jack_1* represent actual components. The arrows between ovals indicate relationships. The *is_a* label indicates an inheritance relationship; properties are copied from the class at the start of the arrow to the class at the arrow's point. The *instance* label indicates that the component at the arrow's point is an instance of the class at the arrows start. The *sub* label indicates that the component at the arrow's point is a sub component of the component at the arrows start.

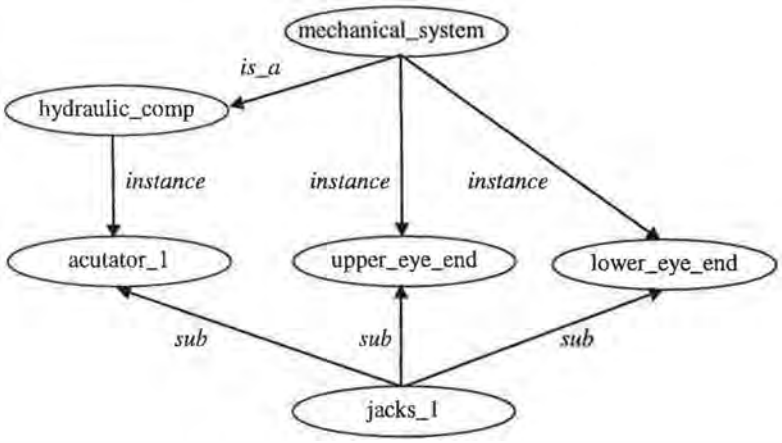


Figure 2-15, Model resulting from the configuration process

The model synthesised by the configuration process details the components of a specific product together with their attributes and interrelationships. The developers of the technique hypothesised that these structures may be used to automate the generation a plan for constructing the product configured (Marshall 1988), thus, developing the MBP technique outlined below. As the emphasis of the research from which MBP was developed was automated configuration, the planning function is not described in detail. Hence, the description of the technique below details only the essence of MBP and not the detailed algorithms required too achieve the functionality. Part of the contribution of the integrated architecture developed in Chapter 7 is the specification of the implementation details of MBP.

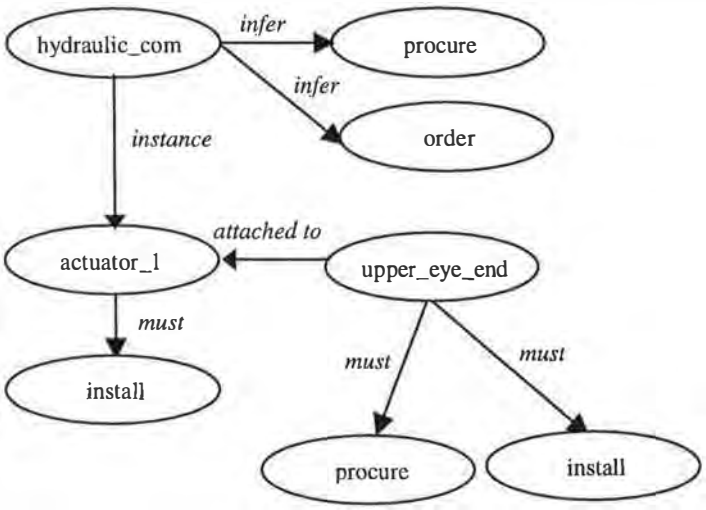


Figure 2-16, Action attachment and dependency relationships

Figure 2-16 depicts the fragment of the flight simulator configuration in Figure 2-15 with action and dependency relationships modelled. The class *hydraulic_comp* is related too two actions *procure* and *order* through the relationship *infer*. This structure specifies that if an instance of the class *hydraulic_comp* is present within a design, then the MBP must determine if the related actions are required.
The *actuator_1* component is an instance of the class *hydraulic_comp* and therefore inherits the *procure* and *order* actions. In addition to the inherited

actions, the component is related to an *install* action through the *must* relationship. This relationship implies that if an *actuator_1* component is present in a design, an *install* action will always be required. The *procure* and *install* actions related to the *upper_eye_end* component are related through the *must* relationships, hence, they must always be included within a plan for a product which contains an *upper_eye_end.*

The MBP assesses each component within a product to determine the actions that should be associated and therefore included within a plan to construct the product. In the case of the *must* relationship, actions are automatically added to the plan. With the *infer* directive production-rules written by the domain modeller are invoked to determine if the action attached should be associated. Two example rule-sets are depicted in Figure 2-17 below.

```
rule-set infer-procure-for-hydraulic-comp
rule-1
 if ?hydraulic-comp.stock-status = in-stock then
      generate procure action
 end-rule
end-rule-set

rule-set infer-order-for-hydraulic-comp
rule-1
 if ?hydraulic-comp.stock-status = out-of-stock then
      generate order action
 end-rule
end-rule-set
```

Figure 2-17, example rule-set for *infer* relationship

The rule-sets will include a *procure* action if the *hydraulic_comp* is *in-stock* or a *order* action if the component is out of stock. The actions synthesised by a MBP for the model instance in Figure 2-16 are depicted below.

```
procure actuator_1
install actuator_1
procure upper_eye_end
install upper_eye_end
```

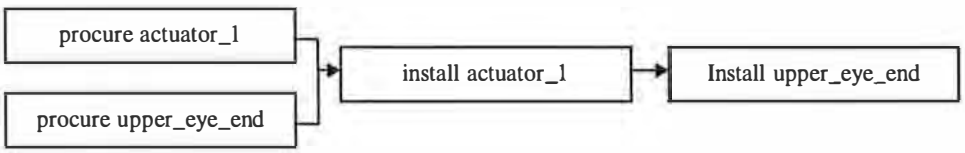
With action synthesis complete, the MBP considers the dependency relationships between actions. Two types of dependency knowledge may be specified. First, the domain writer may specify production-rules for ordering a components own actions. Within the *actuator-1* component the domain writer would specify that if either a *procure* or *order* action is required, the action must be ordered before the install action. The actions above are depicted below after the assessment of ordering constraints local to each component.

procure actuator_1	: pre {∅}
install actuator_1	: pre {procure actuator_1}
procure upper_eye_end	: pre {∅}
install upper_eye_end	: pre {procure upper_eye_end}

The second dependency assessment mechanism considers the relationships between components. In Figure 2-16 the *upper_eye_end* component is *attached to* the *actuator_1* component. The domain writer may attach production-rules to this relationship that determine the ordering constraints which should be added to the actions of the components related through the relationship. In the *attached to* case, the rules specify that the *install* action of the *actuator_1* component must be completed before the *install* action of the *upper_eye_end* component. The actions and ordering constraints above are reproduced below with the ordering constraints resulting from the relationship dependency process included (in bold).

procure actuator_1	: pre {∅}
install actuator_1	: pre {procure actuator_1}
procure upper_eye_end	: pre {∅}
install upper_eye_end	: pre {procure upper_eye_end, **install actuator_1**}

The resultant plan is depicted graphically below:



Whilst MBP successfully determines the actions and ordering constraints based upon domain knowledge encoded in production-rules, the technique does not consider action preconditions and effects. MBP can not, therefore, detect and resolve action interactions. For example, the *procure acutator_1* action may delete the effects of the *procure upper_eye_end* action. Hence, the actions should be ordered so the *actuator_1* action is completed before the *upper_eye_end* action. Without the modelling of action preconditions and effects and the inclusion of an action interaction detection and resolution strategy it is not possible to detect issues of this type.

2.6 Summary and conclusions

The development of classical systems has formed the predominate direction in automated planning research. Classical work has accepted a number of simplifying assumptions about the requirements of planning applications. Specifically, that the world changes instantly, predictably, deterministically, and within the complete control of the agent performing the planning function.

By controlling the complexity of the planning problems considered, classical research has developed a number of powerful technologies that address important facets of planning applications. Precondition achievement planning is based upon an action representation of preconditions and effects. Plans are constructed by searching the set of actions available in an application domain, and combining actions with effects that achieve the goals of the planning problem with actions which ensure the preconditions of all operators are satisfied in a path leading from the initial world state to the goal world state. Task refinement planning structures a domain into a number of partial plan fragments or task networks arranged into a hierarchy of abstraction levels. The task refinement process is the assembly of these partial plan fragments into a complete and interaction free plan.

Precondition achievement planning is the most general planning technology, placing no constraints on the type of application domain to which it is applied. However, the technology's large search space and lack of domain independent search control heuristics has limited its application in industrial domains. Whilst task refinement planning makes the assumption that a domain may be partitioned into a number of relatively interaction free fragments, this assumption has held in a number of industrial application domains. Hence, the ability of task refinement planning to exploit the structure of a domain has proved an important factor in the technology's industrial success.

Current automated planning research is developing classical planning techniques towards applications that do not conform to the limiting classical assumptions. The ARPA / Rome Laboratory Planning Initiative (ARPI) has provided the largest single funding source in recent years, hence, its bias towards real world domains has been seminal in moving automated planning research as a whole into this direction. Current research is addressing issues ranging from the integration of planning, acting, and replanning, through to developing the expressive power of action representations and the tools required to debug and verify planning applications.

The techniques employed by model-based planners differ from classical planning as the technologies have been developed independently. Model-based planning is centred upon a frame-based domain model of a company's products. Planning is effected in the space of components and relationships within such models. In contrast to classical planning, the interactions between the conditions and effects of actions are not considered. The technology is focused upon capturing the domain specific knowledge used by application domain experts when planning.

Model-based planning has been successfully applied to a number of industrial planning application domains. Hence, pragmatically the technique is worthy of further consideration. Its independence from the research effort applied to classical technology, however, has prohibited the cross fertilisation of concepts between the two technologies.

The status of classical plan work towards the end of the ARPI project is summarised by Stillman and Bonsissone as:

> *ARPI researchers have made significant progress in the development of enabling technologies.... Crisis action planning still presents a challenging set of problems.*
>
> (Stillman & Bonsissone 1996, pp 21)

One member of this challenging set was identified at the AAAI workshop on the comparative analysis of planning systems in 1994:

> *Although encoding expert knowledge is at the heart of HTN planning, there remains a considerable gap to bridge in using expert planning knowledge in our systems*
>
> (reported by Wilkins 1994, pp 69)

Model-based technologies have demonstrated through application an ability to capture and reason with expert planning knowledge. Hence, considering the integration of classical techniques with model-based techniques offers potential benefits to automated planning as whole.

In conclusion, classical planning technology in the form of task refinement planning provides a promising industrial technology. Task refinement planning, however, has not reached its full industrial potential. Whilst the current substantial research effort is considering many important facets of task refinement planning, the previous independence of model-based research is prohibiting the inclusions of the technique's ideas. Pragmatically, model-based planning has achieved industrial application success, therefore, the probability of a beneficial cross-fertilisation of ideas is high.

3. Developing a research workbench

Nothing ever becomes real till it is experienced –
even a proverb is no proverb to you till your life has
illustrated it.
John Keats (1795–1821)

3.1 Introduction

The Chambers Dictionary[1] defines a workbench as "*a bench, often purpose-built, at which a craftsman, mechanic, etc. works*". This chapter defines the need for a planning workbench to support the aims of this thesis, and details its implementation. Two appendices support the description with details of the algorithms implemented (Appendix A), and the test specifications used to verify the workbench's correctness (Appendix B).

In this chapter, only the classical component of the research workbench is described. A model-based planner is constructed within Chapter 7 as part of the integrated architecture's development.

3.2 Need for a research workbench

AI planning research is centred upon demonstrable planner prototypes. Historically, a number of early planning research efforts resulted in an implemented prototype system, e.g. STRIPS (Fikes & Nilsson 1971), ABSTRIPS (Sacerdoti 1975a), NOAH (Sacerdoti 1975b), HACKER (Sussman 1974), WARPLAN (Waldinger 1977), NONLIN (Tate 1977), TWEAK (Chapman 1987), DEVISER (Vere 1983). Today, a major thread of research is either developing a demonstrable system, e.g. UCPOP (Barrett & Weld 1992), O-Plan (Currie & Tate 1991), SIPE (Wilkins 1988), PRODIGY (Minton et. al. 1989), SNLP (Pednault 1988) or enhancing an existing system. This central thread of planning research leads to the question: "why are demonstrable planner prototypes so important?" To answer this question, consider the following definitions of AI in general:

> "*AI is ... aiming to achieve* ***functionality in computers****, which when exhibited by humans, is described as having indicated* ***intelligence****.*" (Brooks 1991).

> "*AI may be defined as the branch of* ***computer science*** *that is concerned with the* ***automation of intelligent behaviour****.*" (Luger & Stubblefield 1993).

> "*... the enterprise of* ***programming computers to reason****.*" (Pratt 1994)

> "*AI is a subdivision of computer science devoted to* ***creating computer software and hardware*** *that attempt to produce results such as those produced by people*" (Turban 1992)

The importance of implemented prototypes in AI as a whole is intrinsically linked with the engineering goal of the field. First, prototypes demonstrate the

[1] *The Chambers Dictionary*. Copyright (c) 1994 by Larousse plc

executability of concepts - indicating how close the field has come to reaching its goals. Second, prototypes provide the laboratory for experimentation, analysis and comparison - the methodology for moving the field closer to its goals.

The motivation for constructing a research workbench as part of the research reported in this thesis is two fold. First, the process of experimenting with a planner prototype is an effective way to become familiar with the issues fundamental to planning systems. Second, the resultant workbench will provide an executing model of planning concepts to support the experimentation from which the integrated architecture proposed in the conclusion to Chapter 2 may be developed.

3.3 Realisation approach

Two considerations influenced the approach to realising a workbench. First, the aim of this thesis is to integrate classical and model-based planning technologies. Classical planners require logic reasoning components, whilst model-based planning requires rule-based and object-oriented mechanisms. Second, the research workbench must support rapid and incremental development. Specifically, the workbench must permit new planning concepts to be integrated as they are developed, either as part of the research project or by other researchers.

Two approaches to realising a research workbench were considered: obtaining and modifying an existing planning system, or developing a new planning system based upon details in the planning literature. Table 3-1, below, summarises the systems available at the time the decision was taken (February 1995). Systems that have since become available are presented as shaded entries at the bottom of the table for completeness. The O-Plan system was not publicly available, and its predecessor Nonlin is a commercial product. SIPE, PRODIGY, and UM-NONLIN formed the set of systems that may be used as the basis of workbench.

Planning System	System Requirements.	Availability	Pragmatics
Nonlin (University of Maryland)	LISP	Freely available from web site	
Nonlin (Original)	POP2	Purchase from University of Edinburgh	No financial resources available to purchase the system.
O-Plan	LISP	Distributed only to members of the ARPI initiative.	From July 1997 freely available for research use.
PRODIGY	LISP	Freely available form web site.	
SIPE	LISP Sun workstation	Freely available from web site	Requires Sun hardware.
UCPOP	LISP C++ (soon to be available)	Freely available	

Table 3-1, Summary of the prototype AI Planning systems available in February 1995

Basing the workbench on an existing planning system offered one key advantage. The results of this thesis may be published as based upon a known planning system. However, three considerations outweighed this advantage.

First, integrating classical and model-based planning requires an environment that can support both logic based systems and object-oriented rule-based systems.

Second, using an existing system does not demand the same level of understanding of planning theory as the development of a planning system. Third, the integration was performed with an industrial collaborator which, as in many industrial environments, was based around an IBM-PC compatible infrastructure.

The combination of modelling tool requirements, anticipated experience which may be gained, and the infrastructure of commercial environments led to the decision to develop a new workbench on IBM-PC compatible hardware using Intellicorps KAPPA-PC. The points leading to the selection of KAPPA-PC are summarised in the table below.

- Object-oriented modelling tools.
- Powerful rule-based system integrated into the object-oriented modelling scheme.
- Excellent developers interface, and debugging tools.
- Potential for integration with existing PC applications through DDE.
- Integration with C and C++ programming languages
- Compiled applications may be distributed freely and executed under any Microsoft Windows compatible environment.

With the approach and environment established, existing planning systems were studied to find the detailed knowledge necessary to implement a planning system. The O-Plan system was chosen as the basis for the workbench as the system provides the most clearly defined architecture in the planning literature. Specifically, the architecture defines the modules from which a planner must provide, interfaces between these modules, and details of what function each module must perform.

3.4 Developing the Classical Workbench

This subsection outlines the design and implementation of the classical workbench. Full descriptions of the underlying algorithms are presented in Appendix A. To introduce the underlying design philosophy behind the workbench implementation, the O-Plan system is described first.

3.4.1 Overview of O-Plan

The O-Plan system provides four components:

- *A domain independent representation formalism* (the Task Formalism) which provides mechanisms for specifying the resources, actions, procedures, entities, and relationships within a specific domain. The constructs are domain independent, i.e. they are designed to be applicable in any planning domain
- *A domain independent task specification formalism,* also part of the Task Formalism, provides the constructs for specifying the objective of a planning problem and the actual entities available to solve that problem. The constructs are domain independent.
- *A domain independent planning engine* applies knowledge about a domain encoded in the Task Formalism to solve a specific task.
- *A logical model* or <I-N-OVA> provides a conceptual model underpinning the three concepts above.

The <I-N-OVA> model defines a plan as a set of constraints which together limit the behaviour of agents to that which is desired (Tate 1995; 1996). By providing a description of a plan's components, the model aims to make it possible for a plan to be manipulated in systems other than the activity planner within which it was generated. For example, a plan may be passed to an execution agent for execution, or another (possibly more specialised) planning system for further refinement.

Within the development of the workbench, <I-N-OVA> is viewed as a specification for the data types manipulated by a planner. The O-Plan planning engine (Currie & Tate 1991) may then be classified as an implementation of the functionality required to process the constraints specified in <I-N-OVA>. The domain specification and task specification formalisms are viewed as the constructs for supplying the O-Plan engine with problem definitions and domain specific knowledge detailing the options available to solve those problems.

The <I-N-OVA> model is described below, before the different software modules of the O-Plan system are introduced.

Tate defines a plan as "*a set of constraints which together limit the behaviour that is desired when the plan is executed*"(Tate 1996). These constraints are organised into three sets: Implied, Node, and detail constraints (Ordering, Variable and, Auxiliary).

Implied constraints[2] represent pending or future constraints that will be added to a plan as a result of handling unsatisfied requirements. In precondition achievement planning, the implied constraints represent the conditions yet to be satisfied, and interactions yet to be resolved. In task refinement planning, the implied constraints represent the set of non-primitive tasks to be refined, and interactions yet to be resolved. Implied constraints may therefore be viewed as a planner's "to-do list". When this list is empty, the planner's task is considered complete. <I-N-OVA> extends this model to permit Implied constraints which cannot be processed by a specific generative planner. These constraints may be left unresolved and passed, with the detailed constraints, to a second system for processing.

Plan entities relate to the actions within a plan, and provide the contextual information for the detailed constraints.

Detailed constraints are divided into three categories. Ordering constraints represent temporal relations between nodes. Variable constraints represent co-designation and non co-designation constraints between variables. Auxiliary constraints include conditions, time, resources, authority, and other application specific knowledge types.

The O-Plan system implements a demonstration planning scenario with three agents: a task assignment agent, a planning agent, and a execution agent. The task assigner specifies the task or goal of a plan (i.e. the initial issues). Once task specification is complete, the planner agent is invoked to synthesise a plan to achieve the plan's goals. The completed plan is passed to the execution agent for execution. In the event of execution problems, the execution agent may pass a plan back to the planner with failures represented as issues. If the goals of the plan are no longer maintainable, the plan maybe passed back to the task assigner. This scenario has proved effective in real world domains.

For the purpose of developing a research workbench, only the planner agent of the O-Plan system was considered. The planner agent is described in a number of reports and papers, and its relationship with <I-N-OVA> is described in (Tate 1993a). The main components of the O-Plan planner agent are briefly described below. Familiarity with the system was obtained from a number of sources: experimentation with the University of Maryland's Nonlin implementation, O-Plan technical reports, experimentation with O-Plan over the World Wide Web, and conversations with Dr Brian Drabble at AIAI (in person and via electronic mail).

[2]also referred to as Flaws or Issues or The Agenda.

- **The Associated Data Structure (ADS)**: The ADS maintains the relationship between plan entities (activities in a planner) and the ordering constraints within a plan. The structure provides the contextual information to which more detailed constraints such as conditions, effects, time, resources etc. may be attached.

- **Knowledge Sources:** Knowledge sources encapsulate the plan modification operations (or planning knowledge) of O-Plan. This modularization facilitates experimentation with different implementations of a knowledge source, and concurrent execution of different knowledge sources.

- **Constraint Managers:** Maintain and support detailed plan constraints. Each constraint type supported within O-Plan has a dedicated constraint manager. The set currently implemented includes: the time point network manager, the TOME / GOST[3] Manager, the Resource Utilisation Manager, the Plan State Variable Manager, and the Authority Manager[4]. Collectively, constraint managers support the system's knowledge sources in maintaining plan information.

- **Other Support Modules:** Other support modules provide a variety of support utilities to the system. The current implementation includes plan visualisation tools, instrumentation tools and event handlers. These components and the constraint managers are collectively referred to as the system's support modules.

- **Controller:** The controller is responsible for selecting outstanding flaws or issues within a plan, and managing their allocation to knowledge sources. A flexible allocation scheme is encoded, allowing the order of flaw selection and the mapping of flaws to knowledge sources to be configured to the requirements of a specific domain.

Collectively, O-Plan and <I-N-OVA> provides the data structures and software code specifications for an AI planning system.

[3] The Table of Multiple Effects (TOME) and The Goal Structure (GOST) data structures originate in NOAH (Sacerdoti 1975b) and Nonlin (Tate 1977) respectively. The structures provide a representation for the underlying condition achievement procedure used in O-Plan (c.f. Chapman's Modal Truth Criteria (Chapman 1987))

[4] Designed but not implemented as of August 1997

3.4.2 Implementing the N (Nodes) and O (Orderings)

Within <I-N-OVA>, the node constraints provide the contextual information for the detailed constraints. Issue constraints are processed by modifying the node and detailed constraints. Thus, in terms of implementation, the node constraints provide the most independent part of the model and were therefore implemented and tested first.

The node constraints within <I-N-OVA> model plan entities; in a planner plan entities correspond to activities. O-Plan represents these constraints in a data structure named the Associated Data Structure (ADS) layer. Whilst the <I-N-OVA> model conceptually separates plan entities from their ordering constraints, the ADS is designed to relate them. The structure is confirmed in the following passage from the O-Plan architecture guide.

> *The Associated Data Structure (ADS) provides the contextual information used to attach meaning to the contents of the Time Point Network, and the data defining the emerging plan. The main elements of the plan are activity, dummy and event nodes with ordering information in the form of links as necessary to define the partial order relationships between those elements.*
>
> *(Tate, Drabble & Dalton 1994b)*

The implementation of the ADS within the workbench provides a passive data structure for storing plan context information and the functionality to modify and query this information. The ADS is implemented in three parts: the class *PLAN-NODE*, which represents the individual plan nodes, the class *LINK*, representing the links between nodes, and the class *ADS-MANAGER* which is responsible for maintaining and answering queries concerning the set of plan nodes and links within a specific plan. Each component is described in turn before their interrelationships are defined.

Class PLAN-NODE

Class *PLAN-NODE* records information about an individual activity in a plan. It is derived from the *ALLNODES* data structure in the original Nonlin system (Tate 1976, Page 19), the author's understanding of O-Plan, and the University of Maryland's Nonlin implementation.

The attributes of class *PLAN-NODE* are described in the table below. The class contains no functionality.

Attribute	Description
Predecessors	List of plan node instances which are predecessors of this node. The list contains at least the immediate predecessors.
Successors	List of plan node instances which are successors of this node. The list contains at least the immediate successors
Start Time point	Name of an instance of time point which relates to the start of this activity.
End Time point	Name of an instance of time point which relates to the end of this activity.
Max. Duration	The maximum duration of this node.
Min. Duration	The minimum duration of this node.
Type	Activity | Primitive | Dummy.
Parent	For use within Task refinement planning only. The name of a plan node instance which was replaced by this node during refinement.
Pattern	The pattern of this node. In the form Function arg1..argN.

Table 3-2, Attributes of PLAN-NODE

O-Plan's ADS separates the predecessor, successor, start time, and finish time attributes whilst the workbench's *PLAN-NODE* class integrates them. The O-Plan structure enables the efficient implementation of Operation Research/ PERT type algorithms. Efficiency is not the goal of the workbench, hence, redundant and duplicate data is permitted to enable inspection of plans different perspectives. For details of the O-Plan ADS implementation see Drabble & Kirby (1990).

Class LINK

To support timed delays between activities a link data structure was created. A link represents the relationship activityA < activityB. The start of this link is recorded as the end time point of activityA. The end of this link is recorded as the start time of activityB. Each link records its minimum and maximum duration.

Feature	Description
Start time point	Finish of first plan node.
Finish time point	Start of second plan node.
Min. duration	Minimum duration of the link.
Max. duration	Maximum duration of the link.

Table 3-3, Attributes of LINK

Class ADS-MANAGER

Class *ADS-MANAGER* is responsible for maintaining the set of plan nodes and links within a plan. The class *ADS MANAGER*'s attributes are described in the table below.

Attributes	Description
Nodes list	List of nodes managed - nodes which are not in this list are not part of the plan.
Link list	List of links managed - links which are not in this list are not part of the plan.

Table 3-4, Attributes of class *ADS-MANAGER*

The methods implemented within the class are described in the table below.

Method	Description
Add Node (Pattern, Max. Dur., Min Dur.)	Creates a new node within the ADS.
Add link (Node1 Node2, Min Dur., Max. Dur.)	Add the constraint Node1 < Node2 to the ADS.
Return link Name (node1, node2)	Returns the name of the link between two nodes.
Return nodes code (pattern)	Takes a pattern and returns the node code of the first node found containing the pattern.
Set node type (node, type)	Allows a node to be set as primitive, action or dummy.
Before (NodeX, NodeY)	Returns true if NodeX occurs before NodeY in the plan network.
After (NodeX, NodeY)	Returns true if NodeX occurs after NodeY in the plan network.
Parallel (NodeX, NodeY)	Returns true if NodeX is in parallel to NodeY in the plan network.
Reset	Destroys all information within the ADS, returning a null plan.

Table 3-5, Methods of *ADS-MANAGER*

Querying a plan network to determine how plan entities are related (before, after, in parallel) is a frequent task performed by an AI planner. Fox and Long (1996) describe the efficient algorithm used in their AbNLP (Fox & Long 1995) system. Fox and Long criticise the method employed in many existing planners of dynamically calculating the relationship between plan nodes when answering graph queries. This approach is computationally expensive because the contextual information must be recalculated each time it is required. Fox and Long implement an algorithm which maintains the contextual information, updating it only when new constraints are added to the graph. This approach is more efficient because in general, a plan is queried more often than it is updated. The full algorithm is reproduced from (Fox & Long 1996) in Appendix A.

The overall ADS structure is presented in the object diagram below(Figure 3-1).

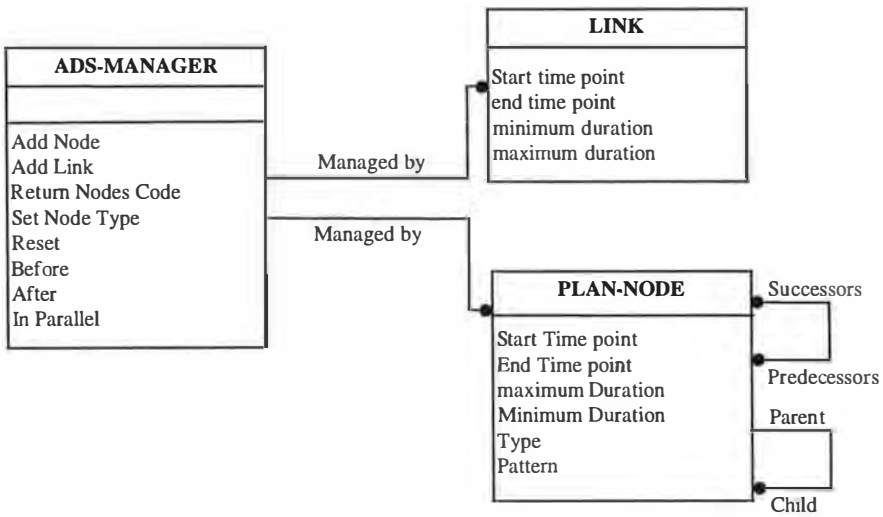


Figure 3-1, Object diagram of ADS system

An instance of class *ADS-MANAGER* manages all the instances of class *LINK* and class *PLAN-NODE* within a plan. Other components of the planner may not modify instances of class *LINK* or class *PLAN-NODE* directly, but must direct their requests through the *ADS-MANAGER*. An instance of the planner workbench will contain one instance of the *ADS-MANAGER*. The one to many cardinality of the "managed by" relationships constrain each *PLAN-NODE* and *PLAN-LINK* instance to be related to one instance of class *ADS-MANAGER* only. It is therefore impossible for a plan node or link to be a member of more than one plan.

3.4.3 Implementing the VA (Variables and Auxiliary)

This subsection describes how the variable and auxiliary constraints are implemented in the workbench, and the construction of plan critics to maintain these constraints.

3.4.3.1 Plan variable relationship critic

The plan variable relationship critic's role is similar to the Plan State Variable Manager's in the O-Plan system. In the plan representation languages used in task refinement planners, variables are not wild cards as in predicate logic, but descriptions of possible instantiations which become further constrained as planning progresses (Wilkins 1988). Variables are declared as being of a specific type, and are therefore constrained to instantiations which are members of this type. The type definitions correspond to enumerated types in procedural programming languages such as ADA and Pascal.

Class VARIABLE

Each plan state variable is represented by the following attributes.

Attributes	Description
Value	The actual value of the variable, if instantiated, otherwise it is set to "none".
Type	Type variable is constrained to be an instance of.
Constraints	A list of co-designation and non-co-designation constraints. These constraints may reference other variables or objects of the variables type.

Table 3-6, Attributes of class *VARIABLE*

Class PLAN-VARIABLE-RELATIONSHIP-CRITIC

Class *PLAN-VARIABLE-RELATIONSHIP-CRITIC* is responsible for creating and maintaining plan state variables.

Method	Description
Create Variable (Type)	Creates a new variable of type specified in the parameter Type, and returns the name of the new variable.
What Is This(Entity)	Replies if Entity is a variable, an object or a type.
Is instantiated(Variable)	Returns true if the variable has been instantiated, false otherwise.
Check co-designation constancy (Variable)	Returns true if the co-designation and non co-designation constraints on Variable are consistent.
Necessarily Co-designate (argument1 argument2)	Returns true if two statements are the same objects, are instantiated to the same objects, or are constrained to be instantiated to the same objects, with the same constraints.
Possibly co-designate (argument1 argument2)	Returns true if two statements unify. i.e. it is consistent to assume they have the same instantiation.

Table 3-7, Methods of CLASS-PLAN-VARIABLE-RELATIONSHIP-CRITIC

The full algorithms for the necessary and possibly co-designation methods are provided in Appendix A. The algorithms are based on the definitions given in (Wilkins 1988, p 72). The architecture of the critique is presented in Figure 3-2 below.

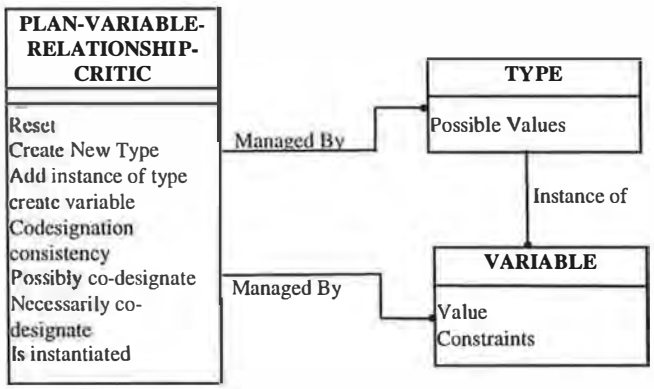


Figure 3-2, Object diagram of plan variable relationship critique system

3.4.3.2 Conditions and effects manager

The conditions and effects manager records the conditions and effects associated with plan nodes (via their time points), and provides the functionality to query and maintain the consistency of these constraints.

Class CONDITION

Class *CONDITION* records the individual conditions in a plan. An entry's contextual information is supported through links to time points. Collectively, the instances of class *CONDITION* make up the Goal Structure of a plan.

Attribute	Description
Type	Condition type: supervised, unsupervised, achieve, only_use_if, only_use_for_query.
Function	e.g. on(block1 block2).
Value	e.g. on | true | red | unset
Time point	Time point at which the condition must hold.
Achieved	Current status: holds, does not hold, may possibly hold with new constraints.
Constraints	Represent the constraints which maybe added to make the condition hold.
Contributor	Time point which contributes to this condition holding.

Table 3-8, Attributes of *CONDITION*

Class EFFECT

Class *EFFECT* records the individual effects within a plan. Collectively, the instances of class *EFFECT* make up the Table of Multiple Effects of a plan.

Attribute	Description
Type	only_use_for_effects or effects
Function	e.g. on (block1 block2)
Value	e.g. True, False, Off, On, Red
Time Point	Time point at which effect occurs.

Table 3-9, Attributes of *EFFECT*

Class CONDITION-AND-EFFECT-MANAGER

Class *CONDITION-AND-EFFECT-MANAGER* supports the creation of conditions and effects, and provides routines to ensure their consistency.

The condition and effect manager includes the Question and Answering functionality of the workbench. The question and answering algorithm responds to question of the type "does statement p hold value v at node n within a plan". The critique responds yes, no, or maybe. The maybe response includes a set of constraints (variable bindings and links) which could be added to a plan to make statement p hold value v at node n. The implementation is based upon the details in (Tate 1976) and reverse engineering of the University of Maryland's Nonlin code. The full algorithm is provided in appendix A.

Method	Description
Add Condition (function, type, at time point)	Creates a condition.
Add Effect (function, type, at time point)	Creates an effect.
QA (P V N)	Responds yes if statement P holds value V at node N. Returns No if the statement definitely does not hold, or may be if it is possible to make the statement hold.

Table 3-10, methods of class *CONDITION-AND-EFFECT-MANAGER*

The architecture of the condition and effect manager is depicted in Figure 3-3 below.

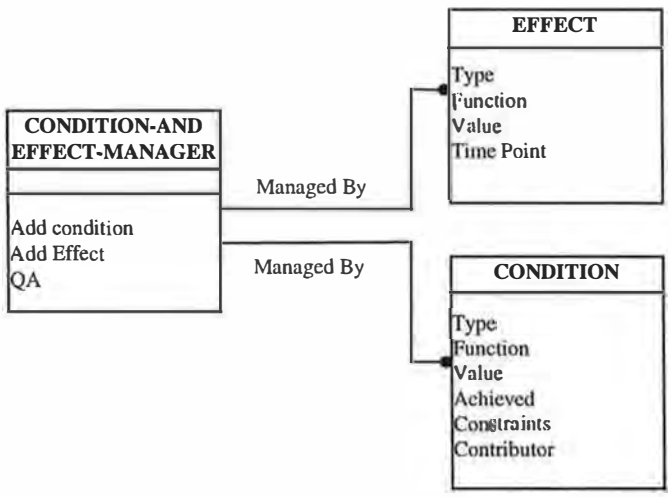


Figure 3-3, Condition and effect manager system

3.4.3.3 Resource and Time Critics

Time and resource critics were not implemented in the workbench. The considerations made for their inclusion at a latter date are described below.

Time critic

Two time points are generated for each activity in a plan to represent each individual activity's start and finish times. Links are provided between activities to allow delays to be specified. Bell and Tate's (1985) constraint maintenance algorithm is implemented to maintain the time information during plan updates.

Vere's (1983) paper describes the algorithms used within the DEVISER system to specify temporal goals using the time point representation implemented. These algorithms have not been implemented in the workbench.

Resources

No resource considerations have been implemented. Conformity to the O-Plan architecture will facilitate the inclusion of a Resource Utilisation Manager and resources attached to time points at a latter date. Algorithms published in (Drabble & Tate 1994) were identified as a potential way of using the resource information to guide search within the workbench.

3.4.4 Implementing the I (Issues)

The Issues component of the <I-N-OVA> model represents the outstanding "to-do list" or agenda of a plan. O-Plan provides a sophisticated mechanism for selecting the next issue to address from the set of outstanding issues. Plan modification operators (e.g. expand a non-primitive task, ensure conditions of a specific type are satisfied) are encapsulated into knowledge sources, which are in turn decomposed into stages. This architecture permits concurrent processing of issues, and a reasoned allocation to be made between the set of outstanding issues and the set of available knowledge sources. Knowledge source stages partition the execution of a knowledge source most commonly into read and write phases. This partitioning allows concurrent reading and writing process to be managed, thus ensuring plan integrity.

Within the workbench, the controller and knowledge sources are integrated into one component, the HTN engine. This architecture is not as flexible as the equivalent O-Plan implementation. The issues surrounding the optimal selection and processing of outstanding flaws are not essential to the aims of this thesis.

The class's methods are described in the table below.

Method	Description
Set Task	Allows the user to select a task from the set of tasks available in the schema library.
HTN Plan (task file, schema file)	The task refinement planning algorithm.
Expand Task(task, schema)	Replaces a non-primitive task with a task network defined in the parameter schema.
Select Method (set of schemas)	Selects a method to replace a non-primitive task, from the set of methods available.
Export plan (filename)	Exports the contents of the ADS into a format accepted by Microsoft Project.

Table 3-11, Methods of class HTN-ENGINE

The HTN Plan algorithm is specified below in Figure 3-4. The algorithm defines how the components of the workbench are controlled to achieve the planning function. A detailed description of the algorithm is provided immediately after Figure 3-4.

```
1.  Read problem definition and task formalism schemas
2.  Append non-primitive tasks from task definition to the task queue
3.  Append achieve conditions from task definition to the task queue
4.  While the task queue is not empty loop
5.          Select a task for processing (either at random, FIFO, or ask user)
6.          if selected task is a non-primitive task then
7.                  -- task reduction procedure
8.                  Ask schema library for methods to achieve selected task
9.                  remove methods whose only_use_if conditions do not
10.                 hold
11.                 if no methods available then
12.                         return to select a task for processing
13.                 end if -- no methods available
14.                 select a method from the set remaining (either at random
                    or ask user)
15.                 expand the selected task with the selected method
16.         else -- it must be an achieve task
17.                 if it is possible to add links to make condition true then
18.                         add links
19.                 else
20.                         ask schema library for schema to make condition
                            true
21.                         if not methods available then
22.                                 return to select a task for processing
23.                         end if -- no methods available
24.                         implement schema immediately before task
25.                         which requires condition
26.                 end if -- it is possible to add links...
27.         end if -- selected task is a non-primitive task
28.         call plan state variable critic
29.         loop until critic replies yes to all variables or user quits
30.         loop
31.                 allow user to correct problems
32.         end loop - PSVM critic
33.         call conditions and effects critique
34.         loop until critic replies yes to all conditions or user quits
35.         loop
36.                 allow user to correct problems
37.         end loop - conditions and effects critique
38. end loop -- while task the task queue is not empty
39. Call PSVC to check unsupervised conditions.
```

Figure 3-4, HTN planning algorithm

The HTN engine is invoked with the parameters "task file" and "schema file". Both parameters refer to physical files on the workbenches hardware platform in ASCII format. The task file contains the set of initial tasks provided by the domain writer, specified in the Task Formalism syntax. The schema file contains the set of non-primitive and primitive task networks specified by the domain writer, specified in the Task Formalism syntax.

Line 1 (of the algorithm in Figure 3-4) instructs the schema library to load and compile the tasks and task networks specified in the task and schema files into the schema library's internal data structures. The user is requested to select a task from the set available to form the objective of the planning process. The HTN engine constructs the initial plan from the task specification.

Line 2 appends all non-primitive tasks in the task definition to the planners task queue.

Line 3 appends all conditions of type *achieve* in the initial definition to the task queue. Achieve conditions may be satisfied in two ways: inclusion of links to make the condition hold, or the introduction of new plan structure to attain the condition.

Line 4 provides the main planning loop's termination criteria. The loop terminates when the task queue is empty. I.e. there are no non-primitive tasks in the network and all achieve conditions have been satisfied.

Line 5 selects a task from the set of outstanding tasks on the task queue. Three methods have been implemented to guide this decision: random, user select or FIFO. The random method selects a task at random. The user method provides the user with the set of outstanding tasks, and asks the user to select the next task for attainment. FIFO (First In First Out) models the method used in the original Nonlin system. Tasks are appended to the end of a queue as they are identified. The planner always selects the first task in this queue for processing next. A system parameter set before planning commences defines which task selection method is used. The task selected at this stage will be referred to as the ***selected tasks.***

Line 6 guards the task refinement path. Only outstanding tasks which are of the type non-primitive task may proceed to the processing functions described in lines 7 - 15 inclusive.

Line 8 (member of task refinement) makes a request to the schema library for methods which are indexed as refinements of the ***selected task***. The schema library returns the set of schemas (possibly null) which match the ***selected task***.

Line 9 (member of task refinement) removes members of the set of schemas returned in Line 8 whose only_use_if filter conditions do not currently hold in the world state. The HTN engine invokes the condition and effect critics question and answering method to achieve this function. Methods whose only_use_if conditions do not attract a yes response are discarded.

Line 11 (member of task refinement) guards the possible case of no methods being applicable to refine the ***selected task.***

Line 12 (member of task refinement) handles the case of no available methods for processing the ***selected task.*** The line returns control to line 5, with the constraint that the next ***selected task*** must not equal the current ***selected task.***

Line 14 (member of task refinement) selects a method from the set of methods available, at random or by questioning the user, for inclusion into the plan.

Line 15 (member of task refinement) replaces the ***selected task*** with the method selected at line 14.

Line 16 is the start of the case where the ***selected task*** is an achieve condition.

Line 17 (member of achieve condition attainment) queries the condition and effect manager's question and answering method to determine if the condition specified in the ***selected task*** is already achieved in the network.

Line 18 (member of achieve condition attainment) if the question and answering method returns yes at Line 17, this line adds a causal link from one of the possible contributors to the ***selected task***. This process is comparable to goal phantomisation in the original Nonlin system.

Line 20 (member of achieve condition attainment) if it is not possible to make the condition specified in ***selected task*** true, this line asks the schema library to return possible schemas for achieving the condition.

Line 22 (member of achieve condition attainment) if no schemas are available for achieving the condition, this line returns processing to line 6, with the constraint that the next ***selected task*** is not equal to the current ***selected task***.

Line 24 (member of achieve condition attainment) expands one of the schemas available to introduce the effect required by the ***selected task***. The schema is placed immediately before the existing task which requires the condition.

The remaining processing is general to both achieve condition attainment and task refinement flaw processing strategies.

Line 28 instructs the plan state variable critic to inspect the variables within the plan.

Line 29 enters the correction loop of the plan state variable critic. The loop terminates either at the users request (the plan may contain inconsistencies) or automatically when all flaws have been processed.

Line 31 displays problems with the plan state variables to the user. The user is invited to add binding constraints to resolve these problems.

Line 33 instructs the plan condition and effect critic to inspect all the conditions within the plan of type achieve, supervised, and only_use_for_query.

Line 34 handles the case of the condition and effect critic identifies unsatisfied conditions, the system enters a correction loop. The loop terminates at either the user's request (the plan may still contain interactions) or automatically when all interactions have been resolved.

Line 36 allows the user to correct interactions by adding variable binding constraints and links to the plan.

Line 38 marks the limit of the main planning algorithm. The line returns control to Line 4 of the program.

Line 39 instructs the condition and effect manager to check unsupervised conditions within the plan. The critic may add links to achieve unsupervised conditions.

The HTN planning algorithm processes two different types of issue: non-primitive tasks and achieve conditions. This functionality allows the workbench to simulate both precondition achievement and task refinement behaviour. Thus, addressing the research constraint raised in Chapter 2 which specifies that the workbench must not be constrained to a particular planning technology.

During the execution of the workbench, an authority relationship is maintained between the HTN Engine and other components of the system. The ADS and plan critics may not modify the plan state. Their interfaces are defined to return constraints which may possibly be added to a plan to resolve conflicts. The components collectively form planning support tools, with the central decision making being made by the HTN algorithm. This authority models the relationships between O-Plan's components and has the advantage of centralising the planner's decision making. Hence, different decision making strategies may be experimented with by modifying one unit of the planning system.

The *HTN-ENGINE* class is supported by a *SCHEMA-LIBRARY* class, which is responsible for reading, maintaining and querying domain knowledge supplied to the planner. Its methods are described in the table below.

Method	Description
Read TF File	Reads a domain description in the Task Formalism format. The schemas and tasks are stored internally for querying.
Return schemas which match (effect)	returns the list of schemas which may be deployed to introduce the effect to the plan.
Return Schemas which match (pattern)	Returns the list of schemas which may be deployed to refine the task described in the parameter pattern.

Table 3-12, Methods of class SCHEMA-LIBRARY

3.4.5 Overall workbench architecture

Figure 3-5, below, presents the component classes of the class *WORKBENCH*.

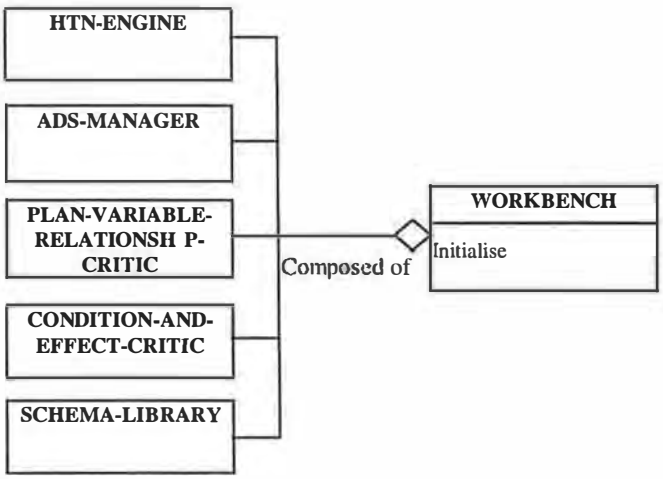


Figure 3-5, Components of the WORKBENCH class

The completed workbench architecture is presented in Figure 3-6 below.

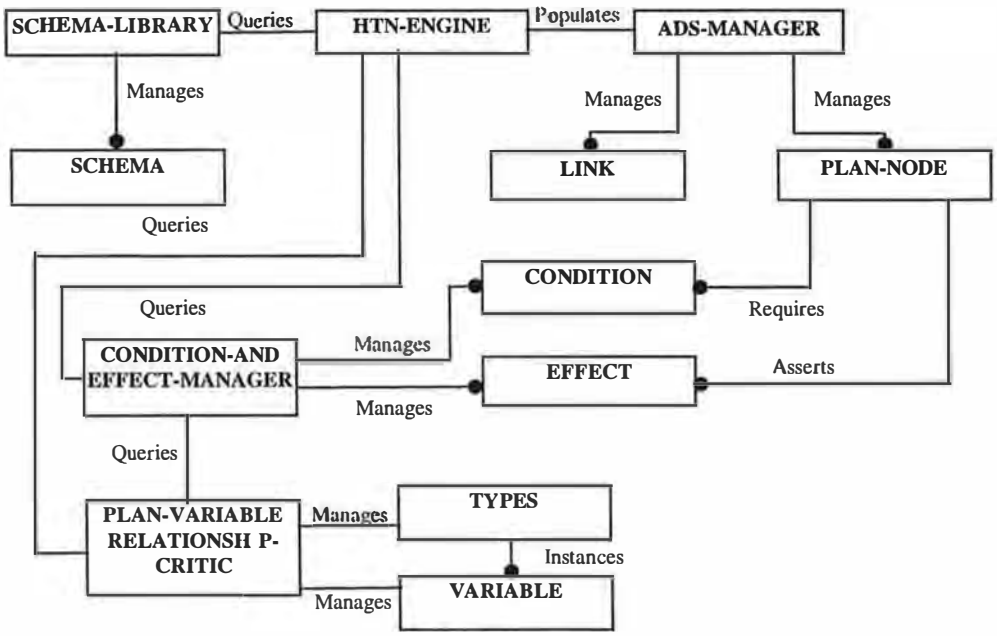


Figure 3-6, Workbench system architecture

This separation of data structures and plan processing functionality facilitates the future development of the workbench. It is possible to replace the O-Plan inspired functionality, whilst leaving the underlying data structures unchanged.

3.5 Implementation details

The workbench is implemented in Intellicorps Kappa-PC version 2.3. This section provides several screen shots to demonstrate the implementation and its user interface.

Figure 3-7 (below) presents the classes implemented within the workbench. The solid lines indicate an inheritance relationship, whilst the dotted lines indicate an instance of a class. Note the inclusion of a TPNM (1)(time point network manager) and a RUM (2) (resource utilisation manager) to permit the extension of the system. The plan depicted is representing a partial solution to the Sussman Anomaly (Sussman 1974). Note the instances of the classes type, movable_objects (3), and objects which have been compiled from the domain representation.

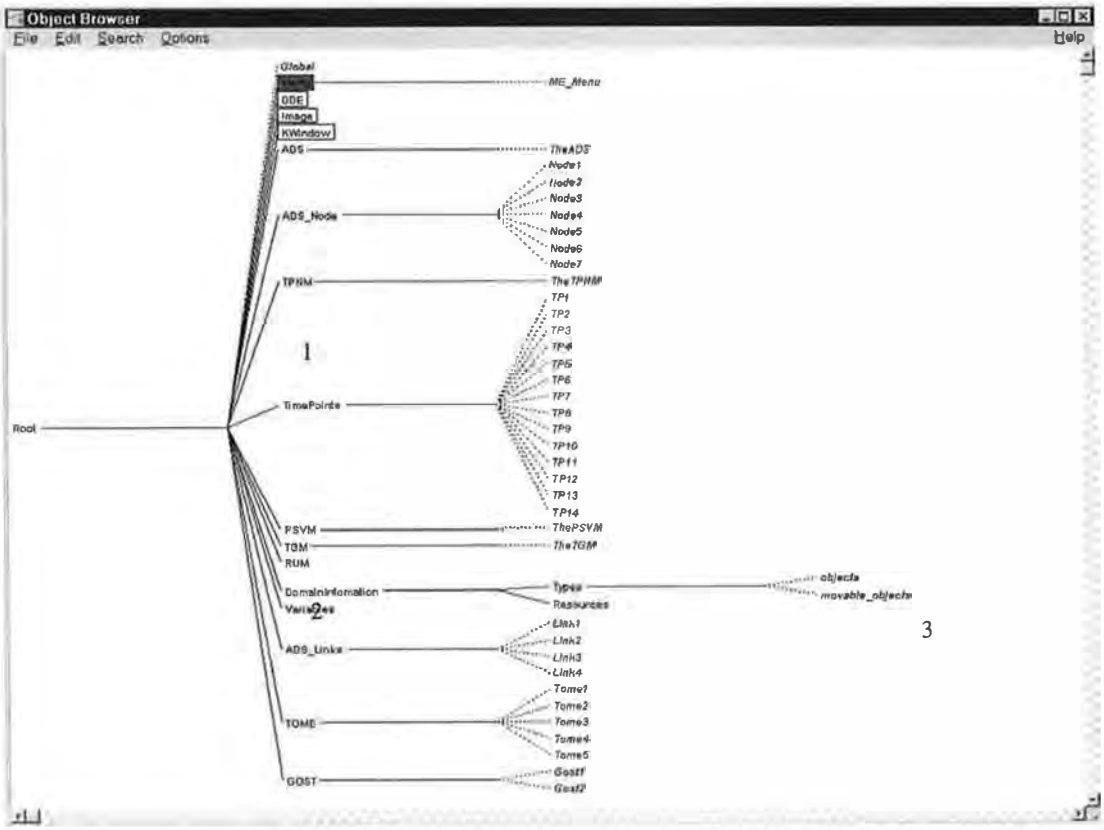


Figure 3-7, Screen shot of the classes implemented in the workbench

The screen shot below (Figure 3-8) expands the TGM (TOME and GOST manager) class in Figure 3-7. to present the classes methods. The QA method is open, revealing a portion of the implemented code.

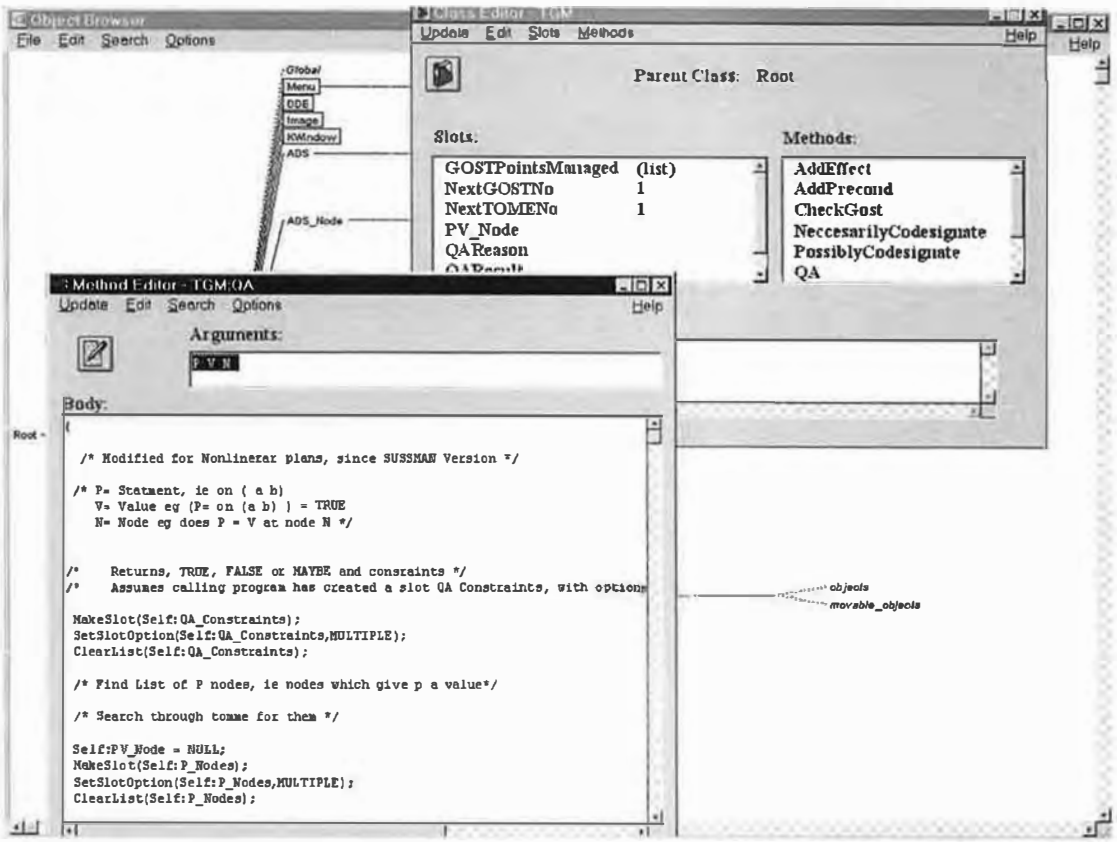


Figure 3-8, TGM class

Figure 3-9 below depicts the user driven interface to the ADS. The interface allows the user to hand code plans without reference to the HTN engine. The interface is split into several windows. The goals and effects of the plan are shown in the two windows at the bottom. The top left window displays the evolving plan and includes the plan modification options open to the user. The top right window is allowing the user to enter a new activity into the plan. This interface was implemented to allow testing of the constraint managers and associated data structure, independently of the HTN engine.

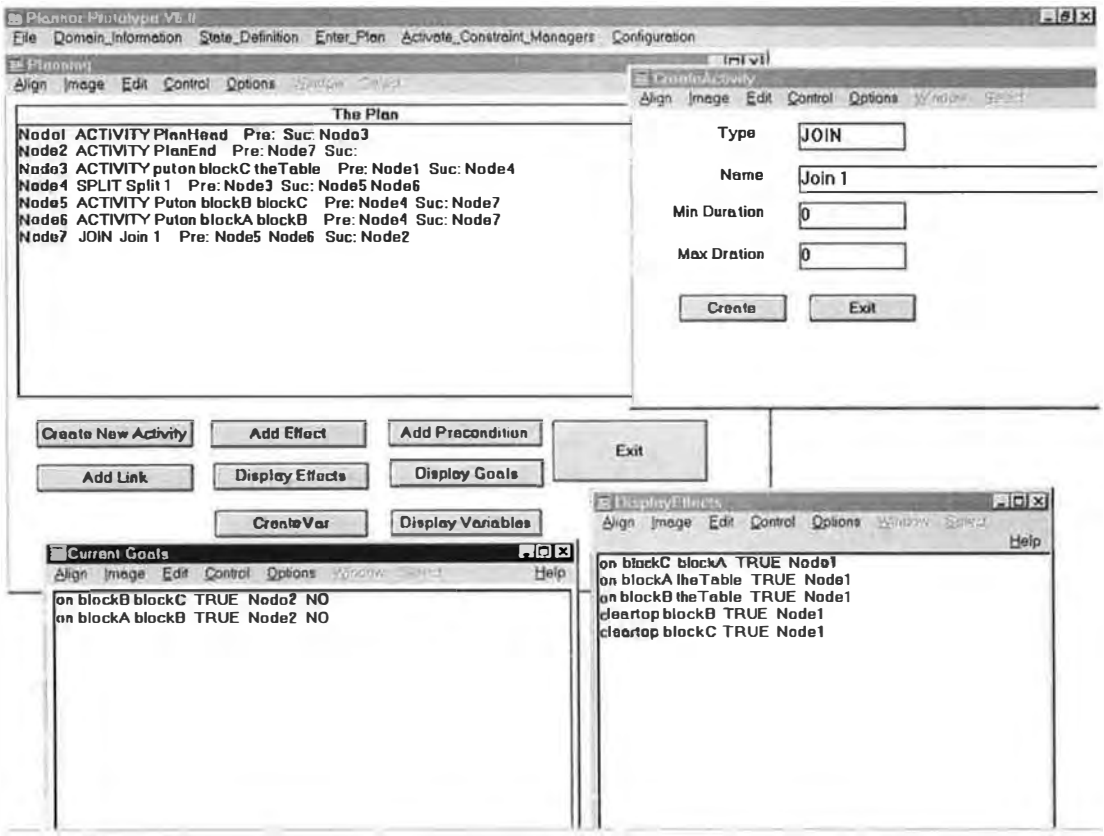


Figure 3-9, Workbench's user interface

3.5.1 Workbench testing

The workbench was tested against two domains from the planning literature: blocks world (Winograd 1971) and Tate's house building domain (Tate 1977). Both domains were encoded using the Task Formalism descriptions from the O-Plan Task Formalism manual. Results were compared against those produced by UM-Nonlin and those reported in (Tate 1976) from the original Nonlin.

Example test cases are provided in Appendix B.

3.6 Summary and conclusions

This chapter identified the role of prototypes in Automated Planning as laboratories to support the field's advancement and demonstrators of the field's current concepts. Within the context of the aims of this thesis, a workbench would form the apparatus for developing the integrated classical and model-based architecture.

A new workbench was developed in an environment which would support logic, object-oriented, and rule-based constructs within the IBM-PC infrastructure of the collaborating organisation.

The planning systems which technically influenced the workbench are depicted in Figure 3-10 below.

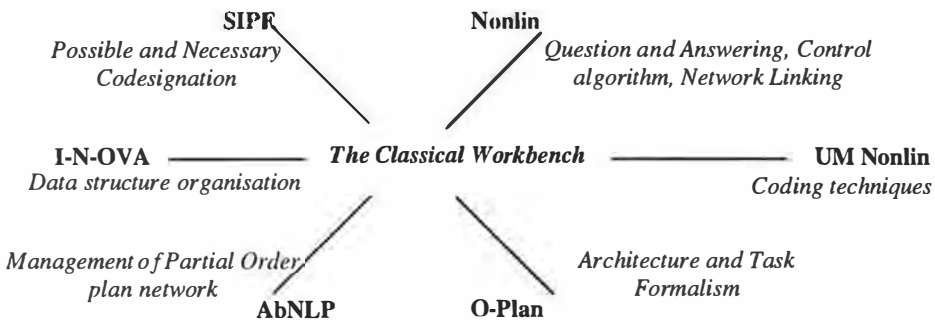


Figure 3-10, Workbench's relationships with existing planner prototypes

In conclusion, the workbench constructed in this chapter is not an automated planning system. The key decision points in the planning process are made by the workbench's user. This strategy makes the operation of the system transparent to its user; hence, allowing the user to view the mechanics of an AI planning system.

The development of the workbench has achieved its aims. First, constructing the workbench has demonstrated to the author the issues encountered when implementing a planning system. Second, it provides a test bed in which a new integrated architecture may be developed and evaluated.

4. Limitations of existing representational devices - experiments within the planning literature

I do not know what I may appear to the world,
but to myself I seem to have been only like a boy
playing on the sea-shore, and diverting myself in
now and then finding smoother pebble or prettier
shell than ordinary, whilst the great ocean of truth lay
all undiscovered before me.
Isaac Newton (1642-1727)

4.1 Introduction

This chapter builds upon the potential for collaboration between classical and model-based technologies identified in Chapter 2 by identifying specific limitations with each technology's representational devices. The limitations are identified through experiments within domains detailed in the automated planning literature. This experimentation is supported by the workbench described in Chapter 3.

Specifically, this chapter aims to identify limitations with the representational devices supported by both classical and model-based planning technologies. Domains from the planning literature are analysed to identify the expert knowledge underlying the current encodings. This underlying knowledge is then modified to include realistic facets of each domain not originally considered. The ability of classical and model-based technologies to represent each modification is assessed.

Before commencing the analysis, the analysis method is described and justified. The method requires an understanding of domains previously considered by planning researchers, hence, an overview of a representative set of these domains is presented. Appendix C supports the overview with the full domain specifications from which it is derived.

4.2 Analysis method

To describe and justify the method used in this chapter to identify limitations with existing technologies, it is necessary to consider the aim of automated planning as a field, and the methods previously employed in its advancement.

Automated planning's aim is to formulate a theory which can simulate[1] planning behaviour. To develop this theory, the facets of planning behaviour must be understood. This transition from behaviour to theory is represented in Figure 4-1 below. The figure depicts the domains of planning theory and planning behaviour with the arrow indicating the transition from the understanding of the facets of behaviour into theory. This transition currently results in the algorithms and domain independent knowledge representation formalisms of planning systems. The question pertinent to this chapter is, "How is this transition achieved?"

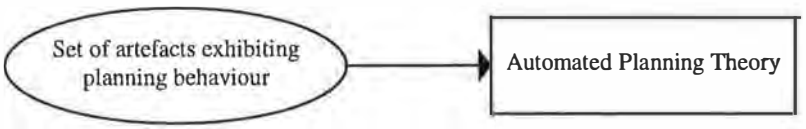


Figure 4-1, Transition from planning behaviour to planning theory

The examination of example problems has been a key method employed in previous work. Ginsberg describes the role of examples in AI in the following passage.

> *The role of examples in AI is to test our theories, and it is the responsibility of individual researchers to conduct an honest search for examples to test their theories... It really is like physics in the sixteenth century; the principal difference is that our experiments are introspective instead of material.* (Ginsberg 1993, p 14)

Ginsberg's observation obscures a significant benefit of working with examples in AI planning. Whilst the use of examples tests the applicability of automated planning theory, ultimately such tests highlight the incompleteness of that theory, and are therefore a driving force in the task of understanding the requirements of planning behaviour. Ginsberg's comparison between AI and sixteenth century physics is pertinent. Automated planning is a relatively young discipline and has yet to establish a significant set of theories upon which research may be built. Hence, it is the process of examining examples that identifies the limitations of the existing theories of planning behaviour.
The research method of formulate example, formulate theory, and formulate counter example is depicted in Figure 4-2 below.

[1] The use of simulate as opposed to emulate is deliberate. The automated planning field does not claim to replicate the actual cognitive processes within humans (Wilkins 1988).

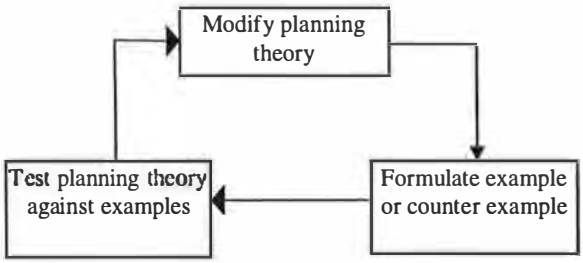


Figure 4-2, Methodology of example and counter example

The methodology of example and counter example is applied in this chapter to identify limitations with existing representational devices. A number of domain definitions in the planning literature are examined to identify the knowledge from which the encoding is derived. With the underlying knowledge identified, the domain description is enhanced to include facets not addressed in the current representation. The ability of existing representational devices to capture these changes in specification and the planning engine's ability to reason with the new knowledge is then analysed.

The next section selects a number of domains from the automated planning literature as the motivating examples for study. With the domains for examination selected, the following sections apply the analysis method described above.

4.3 Overview of a representative set of the domains considered in the planning literature

This section summarises a representative set of domains to which automated planners have been applied. The domains considered are all publicly available; either through the World Wide Web or referenced publications. This qualification excludes some applications of planning technology as commercial applications are proprietary. Hence, it is not possible to obtain them for evaluation.

It is important to note the complexity of modelling application domains in planning formalisms. Chien (1996) states that the complexity of this activity is a prohibiting factor when applying automated planning techniques to industrial planning problems. The issues surrounding this task are discussed in Chapter 5 which describes the elicitation of knowledge from the construction industry.

4.3.1 Blocks world

The blocks world originates in (Winograd 1971) and has provided a motivating problem in planning research (e.g. goal protection (Manna & Waldinger 1974) and action interactions (Sussman 1974)).

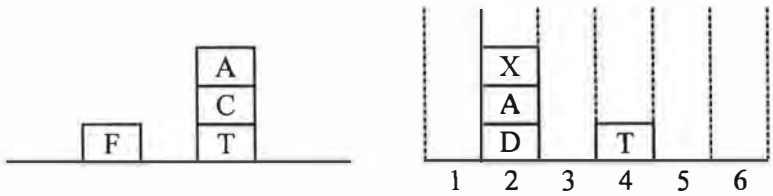


Figure 4-3, Two versions of the blocks world domain

The blocks world consists of a number of blocks and a table (Figure 4-3). The table may be infinite (left side of Figure 4-3) or divided into a number of finite positions (right side of Figure 4-3). The latter modelling adds the complexity of considering where blocks should be placed on the table. Domain knowledge pertinent to planning maybe summarised as follows:

> The world consists of a number of blocks, together with a table. The table may hold any number of blocks (assume the simpler modelling[2]). A block may be moved if it is not obstructed by another block. Only one block may be moved at any given time. The table cannot be moved.

The UCPOP representation of the blocks world's *puton* operator (Barrett & Weld 1992) is depicted in Figure 4-4 below.

[2] The inclusion of positions effectively adds blocks the problem which cannot be moved. Specifically, the planner may move blocks to a position on the table, but may not move that position. This consideration is not essential to the aims of this chapter, therefore the simpler representation is described.

```
1. (define (operator puton)
2.     :parameters  (?X ?Y ?Z)
3.     :preconditions     ( and (on ?X ?Z) (clear ?X) (clear ?y)
4.                 (neq ?X ?Y) (neq ?X ?Z) (neq ?X ?Y)
5.                 (neq ?X Table))
6.     :effect          ( and (on ?X ?Y) (not (on ?X ?Z))
7.                 (when (neq ?Z Table) (clear ?Z))
8.                 (when (neq ?Y Table) (not (clear ?Y)))))
```

Figure 4-4, UCPOP blocks world operator specification

Line 1 and line 2 name the operator and define its parameters. The parameters hold the following semantics:

- ?X : the block to be moved.
- ?Y : the location the block (?X) is to be moved to.
- ?Z : the location upon which the block (?X) is currently positioned.

The operator may be read as *put a block (?X) onto location (?Y) from its current location (?Z).*

The operators preconditions represent two categories of knowledge. Line 3 stipulates that this operator may only be applied if the block to be moved is clear (i.e. no block is on top of it) and the location the block is to be moved to is also clear. Lines 4 and 5 capture relationship knowledge between the operators parameters. The *neq ?X ?Y* constraint stipulates that the block to be moved cannot be the same as the location the block is to be moved to. The *neq ?X ?Z* constraint stipulates that the blocks initial location must not equal its destination location. The *neq ?X ?Y* constraint stipulates that the block to be moved must not be the same as the destination location. The *neq ?X Table* constraint stipulates that the table cannot be moved.

Operator Puton's effects are split into two groups. The effects described on Line 6 always occur as a result of applying the operator. They specify that the block has moved from the initial location (*(not (on ?X ?Z))*), and that the block is at the goal location (*(on ?X ?Y)*). Lines 7 and 8 specify the conditional effects of the operator. If the destination location of the block (*?Z*) is not the table, then the effect that the destination location is no longer clear is asserted (*(when (neq ?Z Table) (clear ?Z))*). If the initial location of the block was not the table, the effect that the initial location is clear is asserted (*(when (neq ?Y Table) (not (clear ?Y)))*).

Consider the representation of the blocks world taken from the O-Plan system (Tate, Drabble, & Dalton 1994b) in Figure 4-5 below.

```
1. always {cleartop TheTable};
2. types   objects        = {BlockA BlockB BlockC TheTable},
3.     movable_objects = {BlockA BlockB BlockC};

4. schema puton;
5.     vars ?x = ?{type movable_objects}
6.          ?y = ?{type objects}
7.          ?z = ?{type objects}

8.     var. relations
9.     ?x /=?y, ?y/= ?z, ?x/=? Z
10.    expands {puton ?x ?y}
11.    only_use_for_effects
12.         {on ?x ?y}        =   true
13.         {cleartop ?y}     =   false
14.         {on ?x ?z}        =   false
15.         {cleartop ?z}     =   true
16.    conditions
17.         only_use_for_query        {on ?x ?z}
18.         achieve                   {cleartop ?y}
19.         achieve                   {cleartop ?x}
20.end-schema
```

Figure 4-5, O-Plan representation of the blocks world

Lines 2 and 3 group the domain entities into two sets, movable objects and objects. The movable object set is a subset of the objects set. The allocation of domain entities to sets captures the knowledge that the table cannot be moved.

The vars section describes the parameters used within the operator. The parameters hold the following semantics:

- ?X : the block to be moved.
- ?Y : the location the block (?X) is to be moved to.
- ?Z : the location upon which the block (?X) is currently positioned

The type constraints upon the variable *?x* prevent the object to be moved being instantiated to the table, as it is not a member of *movable_objects.*

The *var* relations specify the co-designation and non co-designation constraints between the variables. The *?x /=?y* constraint stipulates that the block to be moved must not equal the location to which the block is to be moved to. The *?y/= ?z* constraint stipulates that the location the block is to be moved to must not equal the blocks current location. The *?x/=? z* constraint stipulates that the block to be moved must not equal the location to which the block is to be moved to.

The *only_use_for_effects* section defines the effects of the operator for which this operator maybe used to achieve. The inclusion of an "effects" section would describe side effects of the action. In the puton example there are no instances of effects. The main effects of the action are as follows: *(on ?x ?y) = true* states that the block is now located at the destination location. *(cleartop ?y) = false* states the destination location of the block is no longer clear. *(on ?x ?z) = false* states that the block is no longer on its initial location. *(cleartop ?z) = true* states that the initial location of the block is now clear.

The conditions section defines the conditions of the operator. The *only_use_for_query* prefix denotes variable binding conditions, whilst the *achieve* prefix denotes conditions the planner may add new plan structures to achieve. *only_use_for_query (on ?x ?z)* places a condition that before the operators execution, *?z* must be bound to the object upon which *?x* is located. This condition maybe rebound at any time to ensure its maintenance. The *achieve (cleartop ?y)* and *achieve (cleartop ?x)* conditions specify that the location the block is to be moved to, and the block to be moved, respectively must be clear.

The O-Plan representation encodes the knowledge that the table is infinite through the *always (cleartop TheTable)* at line 1. If the puton operator is instantiated with the destination *TheTable*, the effect *cleartop TheTable = false* will be overridden by the always constraint.

The representations differ slightly in structure due to the different planning knowledge encoded in the UCPOP and O-Plan systems. Negating these differences in representation, both formalisms capture and employ the same knowledge about the blocks world. Both formalisms capture all domain knowledge pertinent to planning.

The essence of blocks world planning problems is identifying and sequencing the movement of blocks to achieve their rearrangement. The successful sequencing of the blocks movement relies on the ability to detect and resolve interactions between actions.

4.3.2 Office world

The Office World originates in the defining STRIPS planning system paper (Fikes & Nilsson 1971). The domain reflects the STRIPS systems initial target domain of robot planning tasks. The world is depicted in Figure 4-6 below.

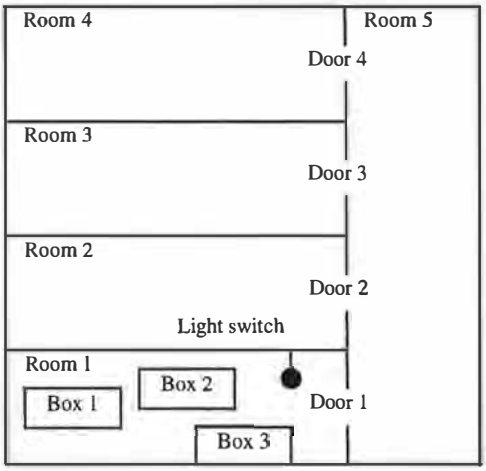


Figure 4-6, The Office world (Fikes & Nilsson 1971), reproduced from (Allen, Hendler & Tate 1990, p 95)

The operators available in the domain are depicted in Figure 4-7 below. The domain consists of a number of interconnected rooms. The rooms contain boxes and light switches. To operate a light switch the robot must be standing on a box, and the box must be located next to the light switch. The robot is free to move boxes around rooms, and from room to room. The constraints on the movement of boxes are that the rooms are connected. For example, the robot cannot move a box from Room 1 to Room 4, without first going through room 5.

Example tasks within the domain include locating boxes in specific rooms and switching on lights.

The office world requires the generation and sequencing of actions to achieve tasks. The domain requires the planner to include actions which achieve the intermediate steps of actions.

```
1. gotol(m) : robot goes to co-ordinate location m
2.     preconditions:
3.         (onfloor) ∧ (∃ x) [in room (robot, x) ∧ locinroom(m, x)]
4.     del list:
5.         atrobot(s), nexto(robot, s)
6.     add list:
7.         atrobot(m)

8. goto2(m) : robot goes next to item m
9.     preconditions:
10:        (onfloor) ∧ {(∃x)[in room (robot, x) ∧ in room (m, x)] ∨
       (∃x)( ∃y) [in room(robot, x) ∧ connects(m, x, y)]}
11.    del list:
12.        at robot (s), next to (robot, s)
13.    add list:
14.        next to (robot, m)

15. pushto(m, n) : robot pushes object m next to item n
16.    preconditions:
17.        pushable(m) ∧ onfloor ∧ next to (robot, m) ∧ {(∃x)
       [in room (m, x) ∧ in room (n, x )] ∨ (∃x, ∃y)[ in room
       (m, x) ∧ connects (n, x, y)]}
18.    del list:
19.        at robot (s), next to (robot, s) next to (s, m), at (m, s)
20.        next to (m, s)
21.    add list:
22.        next to (m, n), next to (n, m), next to (robot, m)

23. turnonlight(m) : robot turns on light switch m
  preconditions:
      {(∃n) [type (n, box) ∧ on(robot, n) ∧ nexto(n, m) ∧
        type (m, lightswitch)] }
  del list:  status (m, off)
  add list: status (m, on)

24.climeonbox(m) : robot climbs up on box m

25. climeoffbox(m) : robot climbs off of box m

26. gothrudoor (k, l, m) : robot goes through door k from room l into room m
```

Figure 4-7, Fragment of STRIPS world operator specification

4.3.3 Briefcase domain

The briefcase world originates in (Pednault 1988) and demonstrates the need for both universal quantification and conditional effects in domain operator descriptions. Planning knowledge pertinent to the domain is described in the paragraph below.

> An object may only be put in a brief case if it is at the same location as the brief case. When objects are taken out of the briefcase they are located at the current location of the briefcase. If a brief case moves, all objects inside the briefcase moves.

The operators available in the briefcase domain are described below (Figure 4-8).

```
1. define (operator move-b)
2.     :parameters (?m ?l)
3.     :precondition (and (at B ?m) (neq ?m ?l))
4.     :effects  (and (at b ?l) (not(at B ?m)))
5.         (forall (?z)
6.          ( when (and (in ?z) (new ?z B))
7.              ( and (at ?z ?l) (not (at ?z ?m)))

8. define (operator put-in)
9.     :parameters(?x ?l)
10.    :preconditions (neq ?z  B)
11.    :effect (when (and (at ?x ?l) (at B ?l))
12.        (in ?x))

13. define (operator take-out)
14.    :parameters (?x)
15:    :preconditions (neq ?x B)
16.    :effect (not (in ?x))
```

Figure 4-8, Briefcase domain representation (UCPOP)

The *move-b* operator specifies the relocation of the briefcase from place *?m* to place *?l*. The action's preconditions are similar to those in the blocks world. The briefcase *B* must not equal to location from which the briefcase is being moved. The destination location *?l* must not equal the initial location *?m*. The effects demonstrate the use of universal quantification in an operators effects. The *forall (?z)* component specifies that for all the entities *?z* which are in the briefcase, the location of the set *?z* moves to location *?l* with the briefcase.

The *put-in* operator describes the action of putting objects into the briefcase. The conditional effect *when at (?x ?l)* and *at(?b ?l)* specifies that the object being put into the briefcase must be at the same location as the briefcase for the effect *(in ?x)* to be asserted.

A typical planning task within the domain would involve the movement of objects to a variety of locations, using the briefcase as the method of transport.

The briefcase domain demonstrates the need for universal quantification and conditional effects in domain operators. The planning complexity is similar to the blocks world and the office world domains.

4.3.4 Tate's house building domain

The house building domain originated as a test domain within the Nonlin project (Tate 1976). It is currently used to demonstrate the Task Formalism domain representation language and the operation of the O-Plan system.

The design of the house considered in the domain from which a construction plan is generated is depicted in Figure 4-9 below.

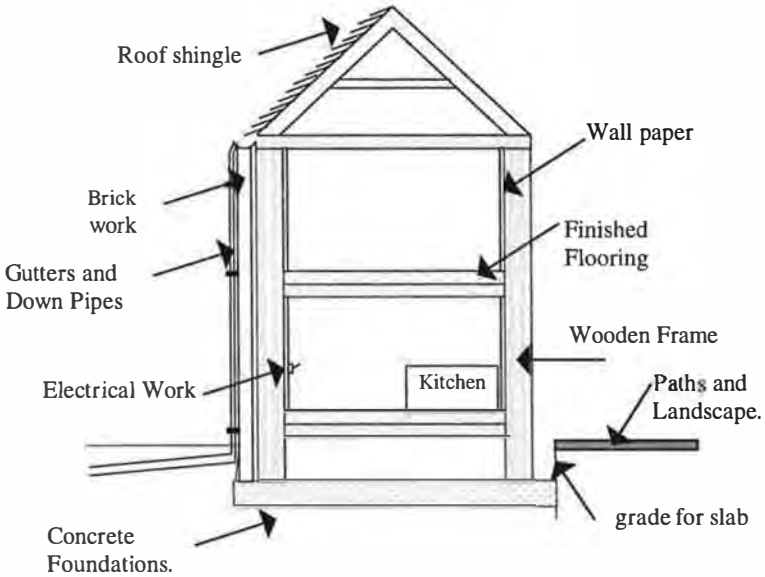


Figure 4-9, Graphical representation of the house in Tate's house building domain

A proportion of the domain's planning knowledge is described in the text below.

The building is made up of a number of interrelated components. The planning task is to assemble these components to achieve the construction of the house. The house has two major components: the foundations, and the walls and roof. The foundations are further decomposed into a number of sub-components: the location the slabs are to be laid, the reinforcement rods, and the concrete slabs. The walls and roof are further decomposed into a number of sub-components: the wooden frame, the exterior sheathing, the insulation, the sheetrock and plaster, the gutters and down spouts, the shingle for the roof, the brickwork.

The foundations must be completed before work commences on the walls and roof because they support the walls and roof. The rough plumbing and wiring must be installed before the outside walls are insulated because the insulation encloses the rough plumbing.

A fragment of the domain representation is depicted in Figure 4-10 below. The modelling of the problem is analysed immediately after this figure.

```
task build_large_house;
  nodes 1 start,
        2 finish,
        3 action {build house};
  orderings 1 ---> 3, 3 ---> 2;
end_task;

schema build;
  expands {build house};
  nodes    1 action {obtain building permit},
           2 action {lay foundations},
           3 action {build walls and roof},
           4 action {joinery},
           5 action {decorate and fit},
           6 action {install services},
           7 action {landscape},
           8 action {close out house};
  orderings 1 ---> 2,  2 ---> 3,  2 ---> 4,  2 ---> 5,
            2 ---> 6,  2 ---> 7,  3 ---> 8;
  conditions unsupervised {wooden frame and roof erected} at 5;
end_schema;

schema lay_foundations;
  expands {lay foundations};
  nodes    1 action {clear lot and grade for slab},
           2 action {place concrete forms reinforcement rods and sewer lines},
           3 action {pour slab};
  orderings 1 ---> 2,  2 ---> 3;
  effects {foundations laid};
end_schema;

schema build_walls_and_roof;
  expands {build walls and roof};
  nodes    1 action {erect wooden frame including roof},
           2 action {fasten exterior sheathing},
           3 action {insulate outside walls},
           4 action {sheetrock and plaster inside walls},
           5 action {place insulation in attic},
           6 action {attach gutters and downspouts},
           7 action {shingle roof},
           8 action {lay brickwork exterior walls plus inside fireplace};
  orderings 1 ---> 2, 2 ---> 3, 3 ---> 4, 4 ---> 5, 5 ---> 6, 1 ---> 7,
     7 ---> 3,  2 ---> 8,  8 ---> 5;
  conditions unsupervised {foundations laid} at 1,
          unsupervised {rough plumbing installed} at 3,
          unsupervised {rough wiring installed} at 3,
          unsupervised {exterior trim complete} at 8;
end_schema;
```

Figure 4-10, Fragment of house building domain representation

The house building domain is modelled at a number of abstraction levels. The first abstraction level is represented by the schema *build.* The schema partitions the house building task into a number of sub tasks.

1. obtain building permit
2. lay foundations
3. build walls and roof
4. joinery
5. decorate and fit
6. install services.

Each task represented at this level may have a set of methods available for its refinement. In this example domain, only one method is available for the refinement of each task. The use of modelling levels permits constraints to be placed at different levels of abstraction. For example, the ordering constraint *1*-->2 is stipulating that the building permit must always be obtained before the foundations are laid. Hence, all the sub tasks of *lay foundations* will be ordered after the sub tasks of *obtainin building permit.* The planning system does not have to deduce this constraint, it must only maintain it.

The unsupervised condtion *wooden frame and roof erected at 5* stipulates that the sub tasks of *decorate and fit* require the wooden frame and roof to be errected before they commence. The use of the unsupervised condtion type constrains the planner to make this condtion hold through ordering constraints only. It is assumed that at atleast one other point in the plan this condtion will be made true.

The second modelling level describes the methods available for attaining each of the tasks described at the first modelling level. One method is avialble for refining the *build walls and roof action*; hence, refining this task will result in the following tasks being added to the plan:

1. erect wooden frame including roof
2. fasten exterior sheathing
3. insulate outside walls
4. sheetrock and plaster inside walls
5. place insulation in attic
6. attach gutters and downspouts
7. shingle roof
8. lay brickwork exterior walls plus inside fireplace

As with the first modelling level, the second level may place constraints upon tasks which all refinements of the constrained tasks must maintain. E.g. the ordering constraint *1*--> *7* captures the constraint that the roof must be laid before the roof covering is installed.
The second modelling level contains a number of unsupervised conditions. The use the unsupervised condition type indicates that the schemas at this modelling

level are aware of a number of conditions required for their attainment, but are unaware of the task or tasks which obtain them.

Effects, like conditions, are modelled at different levels of abstraction. At the second modelling level the lay_foundations action asserts that the effect *foundations laid*. This is a high level effect, describing the aggregation of effects resulting from the tasks possible refinements.

4.3.5 Pacifica

Pacifica (Reece et. al 1993) is an imaginary evacuation scenario developed by the Artificial Intelligence Applications Institute at Edinburgh University to provide a test domain for transportation logistics problems. The domain is summarised in the text below.

> The evacuation problem consists of three tasks: the deployment of evacuation equipment, effecting the evacuation of outlying areas to a central point, the evacuation of people and evacuation equipment from the central point to a safe location.
>
> The deployment of evacuation equipment involves the loading of air and ground transports onto cargo aircraft. The cargo aircraft must fly to location to be evacuated before the evacuation equipment may be unloaded. The aircraft must take off from runways one at a time. Evacuation equipment and cargo aircraft must be in the same physical location before loading may commence.
>
> Effecting the evacuation requires the deployment of ground and air transports to move the people to be evacuated to a central point. A finite number of air and ground transports are available. An air or ground transport cannot perform two trips simultaneously.
>
> After the people to be evacuated have been located at a central point, the evacuation equipment and people are loaded onto transport aircraft and removed from evacuation location.

An example Pacifica task definition (*Operation Columbus*) is depicted in Figure 4-11 below. The task definition is broken down into three sections, reflecting the structure of the domain described above. Tasks 3 and 4 specify the location of evacuation equipment. Tasks 5, 6, and 7 specify the evacuation of the area. Tasks 8, 9 and 10 specify the return of evacuees and evacuation equipment to safety.

In addition to specifying the component tasks of Operation Columbus, the task definitions specifies the current location and status of the area in which the operation is going to be undertaken. The location of evacuation resources and cargo planes is specified in the task definitions effects (e.g. *location_gt GT1 = Honolulu at 1, at C141 Honolulu at 1*). The task definition also specifies the initial runway status at both Delta and Honolulu as clear, and the capacity of the transport air craft and trucks (*gt_capacity 25, at_capacity 35*).

```
task operation_columbus
  nodes sequential
      1 start,
      parallel
          3    action (transport_ground_transports Honolulu Delta)
          4    action (transport_helicopters Honolulu Delta)
      end_parallel
      parallel
          5    action (evacuate Abyss 50)
          6    action (evacuate Barnacle 100)
          7    action (evacuate Calypso 20)
      parallel
          8    action (fly_passengers Delta Honolulu)
          9    action (transport_ground_transports Delta Honolulu)
         10    action (transport_helicopters Delta Honolulu)
      end_parallel
      2 finish
  end_sequenctial;
  effects
      (location_gt GT1) =  Honolulu at 1
      (location_gt GT2) =  Honolulu at 1
      (in_use_for GT1) = in_transit  at 1
      (in_use_for GT2) = in_transit  at 1
      (location_at AT1) = Honolulu at 1
      (in_use_for AT1) = in_transit  at 1
      (apportioned_forces GT)  at 1
      (apportioned_forces AT)  at 1
      (at C141) = Honolulu at 1
      (at C5) = Honolulu at 1
      (at KC10) = Honolulu at 1
      (at B707) = Delta at 1
      (runway_status_at Delta) = clear at 1
      (runway_status_at Honolulu)  = clear at 1
      (gt_capacity 25) at 1
      (at_capacity 35) at 1
end_task;
```

Figure 4-11, Operation Columbus task definition

The encoding of the domain exploits task refinement behaviour for completing the tasks 3, 4, 8, 9, and 10, and precondition achievement behaviour is exploited to complete the tasks 5, 6, and 7. One task from each attainment method is described in detail below. The schema "transport ground transport Honolulu Delta " is used as an example of the methods employed in tasks 3, 4, 8, 9, and 10. The schema *Evacuate abyss 50* is used as an example of the method employed in tasks 5,6, and 7.

Schema transport ground transports

```
schema transport_ground_transports
  expands {transport_ground_transports ?from ?to}
  vars?from        = ?{type air_base}
       ?to       = ?{type air_base}
  nodes
       1 action {load ground_transports}
       2 action {take_off_from ?from}
       3 action {fly_to ?to}
       4 action{land_at ?to}
       5 action {unload ground_transports}
  orderings 1->2, 2->3,3->4,4-5
  conditions
       achieve {at c5} = ?from at 1
       unsupervised {location_gt GT1} = ?from at 1
       unsupervised {location_gt GT2} = ?from at 1
       unsupervised {runway_status_at ?from} = clear at begin_of 2
       supervised {runway_status_at ?from} = inuse at end_of 2 from begin of 2
       unsupervised {runway_status at ?to} = clear at begin_of 4
       supervised {runway_status_at ?to} = in_use at end_of 4 from begin of 4
  effects
       {at c5} = ?to at 5
       {location_gt GT1} = ?to at 5
       {location_gt GT2} = ?to at 5
       {in_use_for GT1} = available at 5
       {in_use_for GT2} = available at 5
       {runway_status_at ?from} = in_use at begin_of 2
       {runway_status at ?from} = clear at end of 2
       {runway_ststus at ?to} = in_use at begin of 4
       {runway_status at ?from} = clear at end_of 4
end_schema
```

The schema encodes the task of locating the ground transports required for an evacuation operation. The parameters *?from* and *?to* specify the initial location from which transportation is to begin and the location from which evacuation is to commence respectively. The typing of these parameters as *air base* prevents missions being considered which fly to or from a location which is not an air base.

The schema decomposes the task of locating the ground transports into the sub tasks *load ground transports*, *take off* from the initial location, *fly to* the location to be evacuated, *land at* the location to be evacuated and *unload ground transports* at the destination location. These actions are ordered in the sequence *load, take off, fly to, land at, unload.*

The conditions *achieve at C5 at 1* permits the planning system to generate plan components which move the cargo plane to the ?from location. The unsupervised constraints of *loacation_GT1 = ?from and location_GT2 ?from* records the assumption that the ground transports will be located at the *?from* location by another part of the plan. The unsupervised *runway status = clear* constraints record the knowledge that the planner should make no attempt to clear a runway. The planner must order the take off and land tasks when the runway is clear. The supervised runway constraints record when the sub tasks of the schema change the status of the runway.

The schema's effects specify that the cargo plane and its cargo of ground transports will be located at the evacuation location when the actions of the schema have been executed.

Schema evacuate city

```
schema evacuate_city
expands   {evacuate city ?city ?number}
vars  ?city          = ?{type city}
  ?number      = ?{satisfies numberp}
conditions
  achieve {evac_status ?city} = {0 ?number};
end_schema;
```

The schema evacuate city provides an expansion for tasks of the form *evacuate city ?city ?number*. The expansion adds a single condition to a plan, *evac_status ?city = 0 ?number*, typed as achieve. The achieve typing permits the planner to add new plan structure whilst attempting to make the condition it prefixes hold. Two schemes within the domain description possibly achieve the evac_status condition: *schema Road_Transport* and *schema Air_ Transport.* Assume the planner chooses the *schema Air_Transport.*

```
schema Air_Transport
  only_use_for_effects          {evac_staus ?from} = {e_left e_safe};
  vars
      ?from          =?{type city}
      ?to            =?{type air_base}
      ?at            =?{type airtransport}
      ?e_left        =?{type numberp}
      ?e_safe        =?{type numberp }
      ?c_left        =?{type numberp }
      ?c_safe        =?{type numberp }
      ?capacity      =?{type numberp }
      ?take          =?{type numberp }
  nodes
      1 action {fly ?take in ?gt from ?from}
      2 dummy
  conditions
      only_use_if  {apportioned_forces AT}
      only_use_if  {evacuate_to ?to}
      only_use_if  {gt_capacity ?capacity}
      compute      {?capacity ?e_left ?e_safe} = {?c_left ?c_safe)
      compute      {- ?e_safe ?c_safe) = ?take
      achieve      {evac_status ?from} = {?c_left ?c_safe) at 2
      unsupervised {location_gt ?AT} = ?to at begin_of 1
      unsupervised {in_use_for ?at} = available at begin_of 1
      supervised   {in_use_for ?at} = ?from at end_of 1 from begin_of 1
  effects
      {in_use_for ?at} = ?from at begin_of 1
      {in_use_for ?at) = available at end_of 1
end_schema
```

The processing of instantiating the schema's parameters subtracts the number of people one air transport may carry, and asserts a new condition of *achieve evac_status city [?previous number safe + ?capacity of at, ?previous number evacuees in danger - ?capacity of at].* Hence, new achieve conditions are added to the plan until sufficient trips to an outlying city are entered into the plan to evacuate all evacuees from a city.

4.3.6 Flight simulator construction

The PIPPA (Marshall 1988) Model-Based planning system has been applied to the domain of flight simulator construction (Marshal, Boardman & Murray 1987). The text below summarises a fragment of the flight simulators construction domain's knowledge.

> The flight simulator industry regulations requires that a feasibility study is completed before a bid for a contract may be initiated. A feasibility study is made up of at most three documents: An engineering report, a cost break down, and a structured design.
>
> An engineering report is required only if a flight simulator of similar specification has not been build by the company before, or the bid is taking place under US air traffic regulations. In the latter case, the structured design for a previous project must be retrieved and updated to match the project currently under consideration. If a new engineering report is written, or an old report modified, the new document must be approved by an internal engineering committee.

The PIPPA representation of the domain is depicted in Figure 4-12 below.

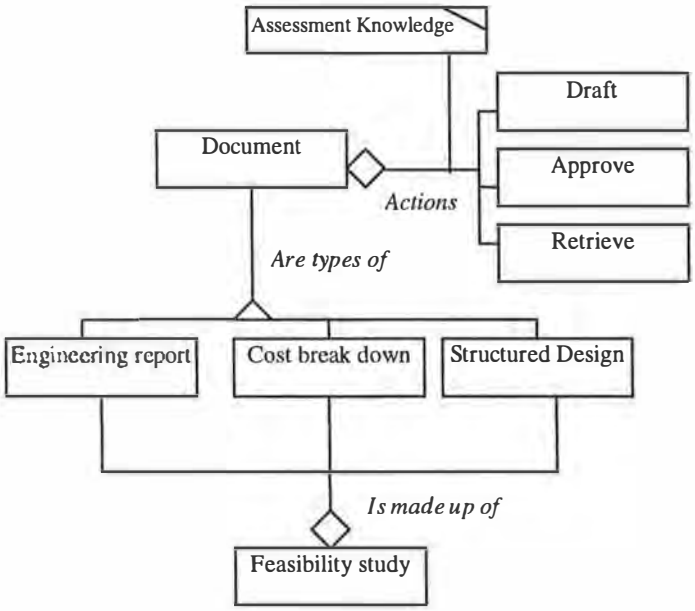


Figure 4-12, Fragment of the PIPPA flight simulator domain representation

The PIPPA model-based representation is centred around the concepts in the domain. In the flight simulator domain, the abstract concept document captures the knowledge that a document may have three actions associated: draft, approve, and retrieve. The assessment knowledge box provides a knowledge marker, indicating that knowledge relating to a specific document type must be inserted to indicate when each of the set of actions should be associated with a specific document.

Engineering reports, cost break downs, and structured designs are represented as specialisation's of the concept document (through the triangle connection in the notation). Each of these specialisations inherit the features of the generic document concept. The feasibility study concept is specified as being possibly made up of the three sub concepts of document.

The feasibility concept contains the knowledge for assessing the need for engineering reports, cost break down and structured design in the form of a rule-based system. An example rule set is depicted below (Figure 4-13).

```
rule #1 requirement for engineering report
if not built similar simulator or standard body = US then
  include a engineering report
end rule

rule #2 decide if built a similar simulator before
if database contains same specs or user identifies similarities then
  built similar simulator = true
end rule
```

Figure 4-13, PIPPA representation of the need for a engineering report

The engineering report concept contains the knowledge for assessing which of the three actions are required to produce it. An example rule set is depicted below (Figure 4-14).

```
rule #3 requirement for draft action on an engineering report in a feasibility study
if standard body = US and built similar simulator then
  retrieve previous engineering report
  approve previous engineering report
end rule

rule #3 not previous built a similar simulator
if not build similar simulator then
  draft engineering report
  approve engineering report.
end rule
```

Figure 4-14, PIPPA action assessment knowledge

Planning proceeds by identifying the actions attached to each component and adding them to the plan.

4.4 Selecting a set of domains from the planning literature

This section examines the current debate within the automated planning concerning the application domains from which further analysis is likely to advance the field. Central to this argument is the future utility of "real world" as opposed to "toy" domains as the motivating examples in planning research. A framework is constructed from this debate to identify domains with a potential utility in experiments to identify limitations with existing representational devices. The resultant framework is then applied to the domains summarised in Appendix C.

4.4.1 Framework for classifying the potential future utility of specific domains

The planning literature contains three perspectives for classifying domains and proposing which offers the greatest utility. Each perspective is summarised below, before a unified framework is constructed.

4.4.1.1 Economical perspective

Drummond (1994) proposes a thesis that precondition achievement planning may be a formulation without significant application. To support this proposition, Drummond provides the following comment on the current domains to which precondition achievement planning has been applied:

> *Of course, one can construct endless toy domains where the operators are specifically formulated so as to facilitate the direct and immediate construction of a solution plan, but such toy domains do not pass our test for economic viability.* (Drummond 1994, pp 3)

Drummond provides the following test for determining the economic viability of a specific domain:

- *Who is the person that wants the problem solved? Call this person the problem's owner.*
- *Does the owner really want a fully automatic solution? Or would they prefer some sort of decision support system - they make the decisions, the system tracks the details.*
- *If the owner really wants an automatic solution, how much are they willing to pay for the required system?*
- *Are there people who are already good at solving the problem, or is it a problem without an existing manual solution?*

(Drummond 1994, pp 2)

Whilst Drummond does not explicitly criticise the use of "toy" domains, he proposes that research should concentrate on better understanding the planning techniques which have succeeded in economically viable applications. He is implicitly assuming the aim of automated planning is to create problem solving tools which may be applied to commercial problems. Drummond's recommendation of focusing on planning technologies which have demonstrated an industrial aptitude must suggest working within domains which pass his test for economic viability.

4.4.1.2 Philosophical perspective

Brooks proposes that AI research should at each stage "*build complete intelligent systems that we let loose in the real world with real sensing and real actions*" (Brooks 1991a), and that those agents should be "*embodied as mobile robots*" (Brooks 1991b). Etzioni (1993) justifies the use of softbots in AI research by accepting the arguments behind Brooks first proposition and refuting the reasoning behind the second.

Brooks provides an engineering methodology argument to justify the use of real domains and therefore robots as the basis of AI research. As part of this argument, he observes the following danger of working within "toy" domains.

> *with a simplified world... it is very easy to accidentally build a submodule of the system which happens to rely on some of those simplified properties ... the disease spreads and the complete system depends in a subtle way on the simplified world.* (Brooks 1991a)

Etzioni applies Brooks' criticism of simplified worlds to justify the use of softbots through the argument that the domains to which softbots are applied have not been engineered by softbot designers, and therefore are not in danger of the simplification disease.

In summary, the philosophical perspective recommends that the domains considered in planning research should not be engineered by planning system designers. By working with "real world" domains, research will avoid developing systems which are dependant upon the simplifications made in "toy" domains.

4.4.1.3 Metric perspective

Andrews et. al. (1995) provide the following justification for designing a realistic bench mark and development problem for planning systems.

> *... a number of toy domains have been devised to assist in the analysis and evaluation of planning systems and techniques. The most well known examples are "Blocks World" and "Towers of Hanoi". As planning systems grow in sophistication and capabilities, however, there is a clear need for planning benchmarks with matching complexity to evaluate those new features and capabilities.*

Andrews et. al. are observing that planning systems have grown in complexity, but the common bench marking domains have not developed in parallel with the technologies they measure. The implication is that the capabilities of the current planning theory is already beyond the minimum requirements of "toy" domains.

The authors go on to define the "matching complexity" of domains through a comparison of their UM Translog domain with the domain upon which it is based (CMU Transport Logistics (Veloso 1992))

> *UM Translog is an order of magnitude larger in size (**41 actions** versus 6), number of features and types of interactions. It provides a rich set of **entities, attributes, actions** and **conditions** which can be used to specify complex planning problems with a **variety of plan interactions**. The detailed set of operators provides **long plans (40 steps)** with **many possible solutions to the same problem**, and thus this domain can also be used to evaluate the solution quality of planning systems.*

The metrics perspective argues that the domains used to test and develop planning systems should match the capability of those systems. The perspective identifies a set of axes upon which domains may vary, and indicates the position of useful domains upon those axes.

4.4.1.4 Selection framework

The past utility of domains formulated by planning system designers was demonstrated in Chapter 2 of this thesis. By focusing on specific aspects of the planning problem, powerful techniques such as the STRIPS action representations (Fikes and Nilsson 1971) and the question and answering procedure over partial-order networks (Tate 1977) have been developed. As the three perspectives above demonstrate, classical planning systems have already developed features capable of addressing more realistic domains, and working continually within simplified domains carries the risk of developing systems which are dependent upon those simplifications.

The two sets of criteria presented in Figure 4-15 below summarise the current thinking within the three perspectives for identifying a domain's type and determining the potential utility of that domain in future planning research.

Type Criteria
1. Is there an identifiable person, other than an AI planning system designer, who wants the problem solved?

Utility Criteria
2. Is there sufficient domain knowledge available to solve the problem?
3. Is there a significant number of entities and attributes in the domain?
4. Are the plans of a significant length?
5. Are there multiple solutions to the problem?

Figure 4-15, Domain evaluation framework

The *type criteria* is derived from Drummond's economic perspective and Brooks' philosophical perspective. The type criteria differentiates between "toy" and "real world" domains. The *utility criteria* are derived from Drummond's economic perspective and Andrews et. al.'s metric perspective. Collectively, the utility criteria provide a test for determining if a specific domain is of sufficient complexity to facilitate future planning research.

The inclusion of the *type criteria* to differentiate "real world" and "toy" domains is not intended to suggest "toy" domains should not be considered by future research. The intention is that if a domain is classed as a "toy" yet of a high utility, it should be carefully considered to ensure resultant work is not unintentionally based upon the simplified facets of the domain.

The following section applies this evaluation framework to the planning domains summarised in Appendix C to identify a set of domains which may be studied with a high probability of identifying limitations with existing representational devices.

4.4.2 Applying the framework to the domains considered in planning literature

This section applies the unified framework developed in section 4.4.1.4 to the domains summarised in Appendix C. All of the domains considered are published; full references are provided within the summaries in Appendix C.

4.4.2.1 Blocks world

Criteria	*Result*
1	It is not possible to envisage a person prepared to pay for solutions to problems in this domain.
2	Yes. The domain may be reproduced in a typical office environment and the domain's knowledge elicited without the need for a domain expert.
3	The domain contains a small number of entities (blocks and a table). The blocks have simple attributes (clear or obstructed).
4	Plans produced are typically quite small (5 or 6 actions)
5	Yes. There are typically several orders in which blocks may be moved, however, there is a limited set of criteria for evaluating solutions (e.g. the length of plans).

4.4.2.2 Office world

Criteria	Result
1	Yes, there is a demand for systems which can accomplish tasks in an office environment. E.g. delivering mail, and cleaning.
2	Knowledge about the actions available in an office domain are easily elicited without the need for a domain expert.
3	If the domain is expanded to include a real office environment the problem would contain a significant number of entities and attributes.
4	If the domain is expanded to include a real office environment the plans required would be of a significant length.
5	If the domain is expanded to include a real office environment there would be multiple ways of achieving tasks.

4.4.2.3 Briefcase domain

Criteria	*Result*
1	It is not possible to define a person who would be prepared to pay for solutions in this domain.
2	Yes, the domain is accessible to any person without the need for a specific domain expert.
3	No. The domain is limited to several objects and a briefcase.
4	The plans are typically short (2 to 10 actions)
5	Multiple solutions are possible but the criteria for differentiating between them are limited to plan length.

4.4.2.4 Tate's house building domain

Criteria	*Result*
1	Yes. There are a large number of construction organisations which may be prepared to pay for a system which plans in this domain.
2	People are available who are currently very good at solving problems in this domain. However, the domain is not immediately assessable to a researcher. A domain expert is required.
3	There are a large number of entities in the domain, each with a large set of attributes.
4	The plans have the potential to be very long.
5	Yes, solution may vary on several dimensions. E.g. resources used, time to construct etc.

4.4.2.5 Pacifica

Criteria	*Result*
1	Yes. Military organisations have identified the need to plan more rapidly during operations.
2	There are people available who are good at solving problems in this domain. The complexity of the domain requires a domain expert for knowledge elicitation.
3	There is a large number of entities with a large set of attributes.
4	The plans may be of a significant length.
5	Solutions may vary against several criteria. E.g. time, cost, resources utilisation, probability of success etc.

4.4.2.6 Flight simulator construction

Criteria	*Results*
1	Yes. Organisations constructing flight simulators have identified a need to improve the speed of their planning process.
2	People are available who are good at solving the problem. The complexity of the domain requires a domain expert for knowledge acquisition.
3	There are a large number of entities with a large set of attributes.
4	The plans may be of a significant length.
5	Plan solutions may vary against several criteria. E.g. time, cost, resource utilisation etc.

4.4.2.7 Conclusion

The domain evaluation framework defined in section 4.4.1 categorises the type of each domain summarised in Appendix C as follows.

"real world" domains under the type criteria

- Tate's House Building Domain
- Pacifica
- Flight Simulator Construction

"toy" domains under the type criteria

- Blocks world
- Briefcase domain
- Office world

The framework assigns the following utility classifications to the same domains.

High utility domains under the utility criteria

- Tate's House Building Domain
- Pacifica
- Flight Simulator Construction

Low utility domains under the utility criteria

- Blocks world
- Briefcase domain
- Office world

None of the domains classified as being of a high utility are also classed as toy domains. Therefore, no special scrutiny is required of the high utility domains before considering their use in further research.

The following sections (4.5 and 4.6) evaluate the representational devices supported by classical and model-based planners respectively. Classical planners are evaluated against Tate's House Building domain and the Pacifica domain. Model-based planners are evaluated against the flight simulator domain.

4.5 Experiments with classical planning

This section describes experiments based upon encodings of Tate's House Building domain and the Pacifica Evacuation domain in the O-Plan system's Task Formalism notation. O-Plan was selected as the system provides a state of the art implementation of task refinement planning. The experimentation identifies the domain knowledge underlying each domain, and then expands each domain definition to include facets not currently represented. The ability of classical planners to represent and reason with each enhancement is examined.

4.5.1 Tate's house building domain

This subsection identifies the components of a building's design and their interrelationships as the domain knowledge from which Tate's House Building domain representation (as defined in Appendix C) is derived. The Task Formalism's ability to support variations in a building's design is then assessed.

4.5.1.1 Identifying the domain knowledge from which Tate's house building encoding is derived

Tate's House Building representation encodes a specific building's design. An overview of the design derived from the encoding is depicted in Figure 4-16 below. A detailed fragment of this design is depicted i n Figure 4-17. The process from which these figures were derived is described after Figure 4-17.

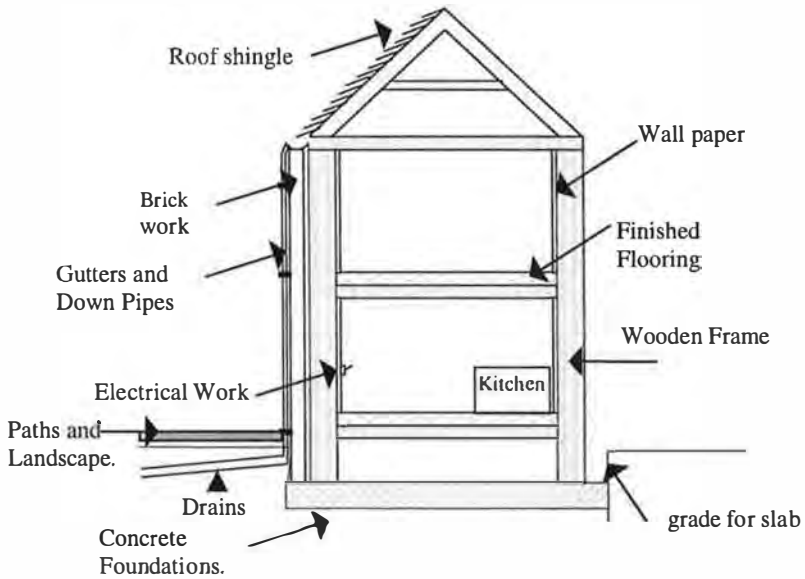


Figure 4-16, Design of the house encoded within Tate's house building domain representation

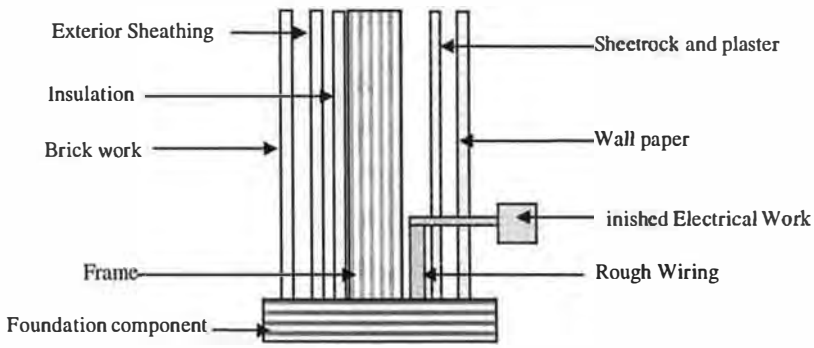


Figure 4-17, Detailed fragment of the house design encoded in Tate's house building problem

- The building's frame is *supported by* the foundation component. This knowledge is encoded in the schemas "build" and "build_walls_and_roof". The schema "build" contains the ordering constraint "2-->3", where action 2 is "lay foundations" and action 3 is "build walls and roof". The schema "build_walls_and_roof" asserts the condition "unsupervised {foundations laid} at 1", where action 1 is "erect wooden frame including roof".

- The exterior sheathing is *attached to* the frame component. This knowledge is encoded in the schema "build_walls_and_roof" through the constraint "1-->2", where action 1 is "erect wooden frame including roof" and action 2 is "fasten exterior sheathing".

- The insulation component is *attached to* the frame and exterior sheathing components. This knowledge is encoded in the schema "build_walls_and_roof" through the constraint "2-->3". Where action 2 is "fasten exterior sheathing" and action 3 is "insulate outside walls"

- The brickwork component *encloses* the exterior sheathing and the outside face of the frame. This enclose exterior sheeting knowledge is encoded in the schema "build_walls_and_roof" through the constraint "2-->8". Where action 2 is "fastening exterior sheathing" and action 8 is "lay brickwork exterior walls plus inside fireplace". The enclose outside face of the frame knowledge is encoded in the constraint "*1*-->2". Where action 1 is "erect frame and roof" and action 2 is "fastening exterior sheeting".

- The sheetrock and plaster component *encloses* the insulation component[3]. This relationship is encoded in the schema "build_walls_and_roof" through the constraint "*3-->4*". Where action 3 is "insulate outside walls" and action 4 is "sheetrock and plaster inside walls".

- The sheetrock and plaster component is *attached to* the inside walls. This relationship is encoded in the schema "build_walls_and_roof" through the constraints "1-->2", "2-->3", "3-->4".

- The rough wiring is *attached* to the wooden frame. This relationship is encoded in the schema "electrical_services" through the constraint "wooden frame and roof erected at 1". Where action 1 is "install rough wiring".

- The finished electrical components are *attached to* the rough wiring and *damage and protrude though* the wallpaper. This constraint is encoded in schema electrical_services. The attached relationship is captured by the ordering "1-->2". And the *damage and protrude through* the constraint "selected surfaces wallpapered at 2".

The components in the fragment of the building considered are summarised in the box below.

foundations, frame, brickwork, insulation, exterior sheathing, sheetrock and plaster, wallpaper, finished electrical work, rough wiring.

The interrelationships between the components of the building fragment are summarised in the box below.

frame	***supported by***	*foundations*
exterior sheathing	***attached to***	*frame*
insulation	***supported by***	*frame and exterior sheathing*
brickwork	***encloses***	*exterior sheathing*
sheetrock and plaster	***encloses***	*insulation*
sheetrock and plaster	***attached to***	*the inside walls*
rough wiring	***attached to***	*the wooden frame*
finished electrical work	***protrudes***	*wall paper*
finished electrical work	***damages***	*wall paper*

[3] It is assumed the insulation is injected through the inside walls into the cavity created by the frame and exterior sheathing.

From the analysis above, it is valid to conclude that Tate's house building domain's encoding is derived from a building's design. With the source of the description identified, the following sections experiment with the ability of classical planning to represent and reason with enhancements to this description.

The house building experimentation defines a hypothetical planning application based upon a construction company which provides a house design which may vary according to a customer's requirements and the area in which it is to be built. The ability of task refinement planning too support this type of application is tested. The experimentation commences by encoding a schema to capture the generic structure of a building. The ability of classical planners to represent the knowledge required to produce a design from this schema based upon a specific design is then considered.

4.5.1.2 Representing the general tasks required to build a house

The generic design of the building provided by the hypothetical construction company may be mapped onto the following Task Formalism schema.

```
schema generic_build;
  expands {build house};
  nodes   1    action {obtain_permission_to_build},
        2    action {lay_foundations},
        3    action {build_structure},
        4    action {carpentry},
        5    action {decorate_and_fit},
        6    action {install_services},
        7    action {landscape};
  orderings
        1-->2, 2-->3, 2-->4, 2-->5, 2-->6, 2-->7;
end_schema;
```

Figure 4-18, Representation of the generic house design's actions

Schema *generic_build* captures the tasks that must be performed for all houses constructed by the building company. The schema is stating that all houses require the tasks *obtain_permission_to_build, lay_foundations, build_structure, carpentry, decorate_and_fit, install_services,* and *landscape*. The ordering constraints state that *obtain permission* to build must always be completed before any other activity and that *lay_foundations* occurs immediately after permission to build is obtained and before any other task. The ordering of the remaining tasks is not constrained.

The task refinement representation proves capable of capturing the generic structure of the house. The following sub sections examine the issues encountered when modelling the more specific levels of abstraction. Semantically, the *generic_build* schema is stating a building will always contain the set of tasks it specifies. It is the responsibility of the lower level tasks to evaluate precisely how each task will be translated into appropriately ordered primitive actions and what those primitive actions should be.

4.5.1.3 Encoding the task *obtain permission to build*

Schema *generic_build* states that a plan to construct a specific building requires actions to obtain permission before building may commence. This section considers the issues encountered when encoding this refinement. The encoding is divided into three cases. Case one considers a single condition (location) affecting the actions required to obtain permission to build. Case two extends case one to consider a number of conditions (historical significance, mining work, footpaths, and sewage) affecting the actions required to obtain permission to build. Case three examines the expressiveness of the *filter condition* construct supported by task refinement planners as a result of the constructs importance identified in cases one and two.

Case 1: Single condition

Assume that permission to build is obtained under the following regulations:

1. If the location of the construction is rural then planning permission is required from the local authority.
2. If the location of the construction is urban then planning permission is required from the local authority and a safety document must be submitted and approved by the local authority.
3. Planning permission cannot be drafted without obtaining an approved safety document in an urban area.

Figure 4-19, Regulations constraining building permission - single condition case

The regulations may be formalised into two methods; one applicable to an urban area, and one to a rural area. This structure is expressed in the figure below.

```
to obtain permission to build
  if location = urban then
      refine using method obtain_permission_to_build_urban
  else if location = rural then
      refine using method obtain_permission_to_build_rural
  end if
```

From the formalisation above, the permission regulations may be translated into the two task networks depicted below. Schema *obtain_permission_to_build_**urban*** encodes the case when building is performed within an urban area. Schema *obtain_permission_to_build_**rural*** encodes the case when building is performed within a rural area. The urban case includes two actions in addition to the rural case: *draft safety schedule* and *submit safety schedule.*

```
schema obtain_permission_to_build_urban;
  expands {obtain permission to build};
  nodes   1   action {draft safety schedule},
        2   action {submit safety schedule},
        3   action {draft planning permission},
        4   action {submit planning permission};
  orderings
        1-->2, 2-->3, 3-->4;
  conditions
        only_use_if area_type = urban;
  only_use_for_effects
        permission to build obtained = true;
end_schema;

schema obtain_permission_to_build_rural;
  expands {obtain permission to build};
  nodes   1   action {draft planning permission},
        2   action {submit planning permission};
  orderings
        1-->2;
  conditions
        only_use_if area_type = rural;
  only_use_for_effects
        permission to build obtained = true;
end_schema;
```

Figure 4-20, Urban and rural refinements for schema *obtain_permission_to_build*

Selection of the appropriate method for obtaining permission to build is achieved through the *area_type* condition that may take the values *urban* or *rural*. The bolded *only_use_if* filter condition in each schema specifies the value of the *area_type* condition under which each method is appropriate. When refining the task *obtain_permission_to_build* the planner will identify both methods as candidate refinements. The planner will then discount the method whose *only_use_if* condition does not hold within the current plan state. Hence, the *_rural* version will be used if *area_type* is set to *rural*, and the *_urban* version will be used if the *area_type* is to *urban*.

This encoding is depicted graphically in Figure 4-21 below. The *generic_build* schema is assigned the modelling level zero, and the schema's two possible refinements level one. Only one of the schemas at level one will be selected as the refinement. Hence, within the figure, task refinement moves downwards selecting one available path at each level.

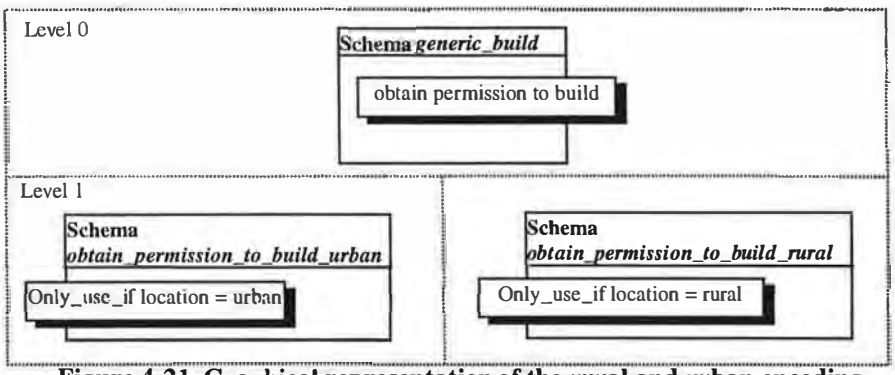


Figure 4-21, Graphical representation of the rural and urban encoding

Generically, the task refinement formalism is representing application domain knowledge of the form:

```
to achieve task t
  If condition = value1 then
      refine using method-a
  else if condition = valuen then
      refine using method-n
  end if
```

The encoding method used to map application domain knowledge of this form into task networks is depicted in Figure 4-22 below. Each method for refining a task is mapped to a task network, and the conditions under which each method is applicable are distinguished by filter conditions.

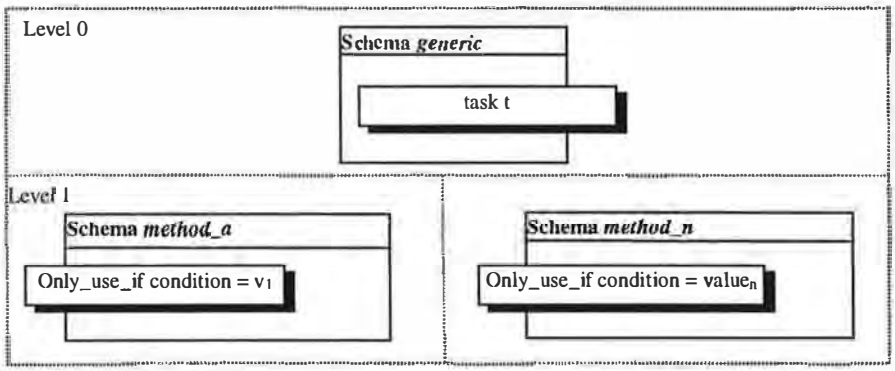


Figure 4-22, Encoding single conditions in the task formalism

This mapping from domain knowledge into a task refinement notation raises a number of issues. First, if a method is encoded for each value a condition may take, as the number of values a condition may take increases, so does the number of methods which must be written. The two methods written for the *obtain permission* to build task contain two common actions (*draft planning permission* and *submit planning permission*). If this knowledge changes, both schemas will require editing. Therefore, as the number of methods encoded increases so does the probability of redundancy in the form of the same action specified in a number of methods. As redundancy grows, so does the maintenance requirement of a domain's specification.

Second, the mechanism for determining the applicability of methods is limited by the expressiveness of the filter condition construct. This issue is considered further in the in case three. The remainder of this case considers only the redundancy issue.

A possible method for addressing the redundancy issue is to identify the actions dependent on and independent from the filter conditions. In the construction example, the regulations constraining permission to build always demand planning permission actions, only the safety schedule actions are dependent upon the location of construction. It is therefore possible to write a schema *generic_obtain_permission* with two sub tasks: *planning permission* and *safety schedule.* As planning permission is always required, only one refinement without filter conditions must be written. The *safety* schedule task, however, requires two possible refinements. The first with a filter condition *location = rural* and the second with the filter condition *location = urban.* The first method will contain one *dummy* action as no safety schedule actions are required in a rural area. The second method will contain the two safety schedule actions as a safety schedule is required in an urban area.

Figure 4-23 below depicts this encoding. Modelling level two contains the two methods for refining the safety schedule task and the single method for refining the planning permission task. The notation separates these sets within a modelling level with a partial dot dash line.

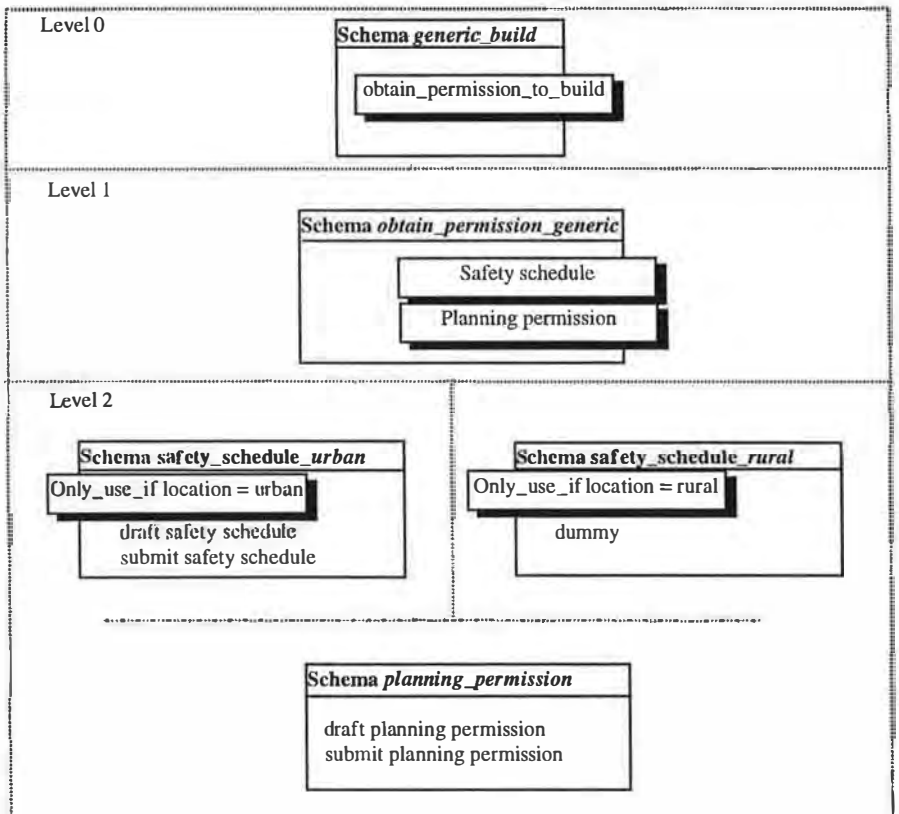


Figure 4-23, Adjusting encoding to isolate the dependant actions

The encoding method presented in Figure 4-23 successfully determines the actions which are required in a given situation whilst addressing the redundancy issue. However, the conditional ordering constraint within the regulations has not yet been considered. Note that line three of the regulations in Figure 4-19 states that a safety schedule must be completed before a planning application is approved.

Within the encoding above, the domain writer may place a *supervised* condition combined with an ordering constraint between the *planning permission* and *safety schedule* tasks at modelling level one. The condition would be of the form *supervised safety schedule submitted at 1 from [2]* (assuming planning permission is node 1 and safety schedule node 2). With this constraint added to the representation, two plans are possible. First, *null --> planning permission.* Second, *safety schedule --> planning permission.* In the first case, the planner will work unnecessarily to maintain the null action before the planning permission action. This encoding therefore addresses the redundancy issue but at the expense of possibly unnecessary ordering constraints for the planner to maintain.

When restricted to the task refinement process, the domain writer cannot use the *unsupervised* condition type at modelling level two to order the safety and planning tasks only in the case when a safety schedule action is required. Supervised conditions may place a precondition constraint only, I.e. the action producing the condition prefixed by the unsupervised type is ordered before the action requiring the condition. Thus, it is not possible for the safety schedule task to constrain the planning permission tasks to follow itself - *unsupervised* conditions work backwards only over a plan.

The single condition case leads to the following conclusion. As the number of values a condition affecting the actions in a plan increases, the number of methods which have to be written increases, and the probability of redundancy between methods will also increase. Whilst it is possible to separate the actions dependant upon a condition from those which are not, such an encoding complicates the specification of ordering constraints. Hence, even in the case where a number of actions may be independent of domain conditions, if ordering constraints between the dependant actions are required, the domain writer must include the independent actions within the dependant action's methods.

Case 2: multiple conditions

Case One considered only a single condition affecting the selection between refinements of a task. This case considers the issues encountered when multiple conditions affect the selection of a task's refinement. For the purpose of this investigation, assume that the original regulations constraining obtaining permission to build in Figure 4-19 are replaced with the following:

1. If the location is of historical significance then permission must be sought from the National Historical Department of the government.
2. If the area contains old mining work then a search must be performed at the local mining companies records office to ensure the location is stable.
3. If the location is crossed by a public foot path then legal advice must be sought to ensure no infringement on public rights of way.
4. If the location contains major serves (water, gas, sewage) then approval must be obtained from the local service providers.

Figure 4-24, Regulations constraining building permission - multiple conditions case

These regulations may be formalised into the following conditions:

historical = true or false
mining area = true or false
public footpath = true or false
sewage = true or false

Figure 4-25, below, presents an encoding of these regulations following the framework developed in Case One above. Each condition is converted into a sub task of the schema *obtain_permission_generic* at modelling level one. A refinement is then provided at modelling level two for the actions dependant upon each condition.

The encoding described in Figure 4-25 assumes that the actions require no ordering constraints. If ordering knowledge is required, the level one schema *obtain_permission_generic* may place ordering constraints and supervised conditions between its four sub tasks. These constraints and conditions must, however, hold for all possible refinements. For example, if *historical significance* must always be performed before the *services* task, an ordering constraint *1-->4* (assume *historical significance* is assigned the label 1 and *services* the label 4) and a supervised condition *supervised historical significance checked at 4 from [1]* may be placed. This constraint and condition will be maintained by all possible refinements of the tasks specified.

If, however, the relationship between the tasks *historical significance* and *services* is not constant, i.e. it varies depending upon the value of conditions in a specific problem instance, the supervised condition and ordering constraint mechanism may not be used. The schemas defined at modelling level two may not place constraints directly on actions in other schemas. For example, schema *historical_significance_true* may not place a supervised constraint on an action introduced by schema *services_true*. Each schema is a unit of encapsulation and supervised conditions and ordering constraints may only be placed within that unit.

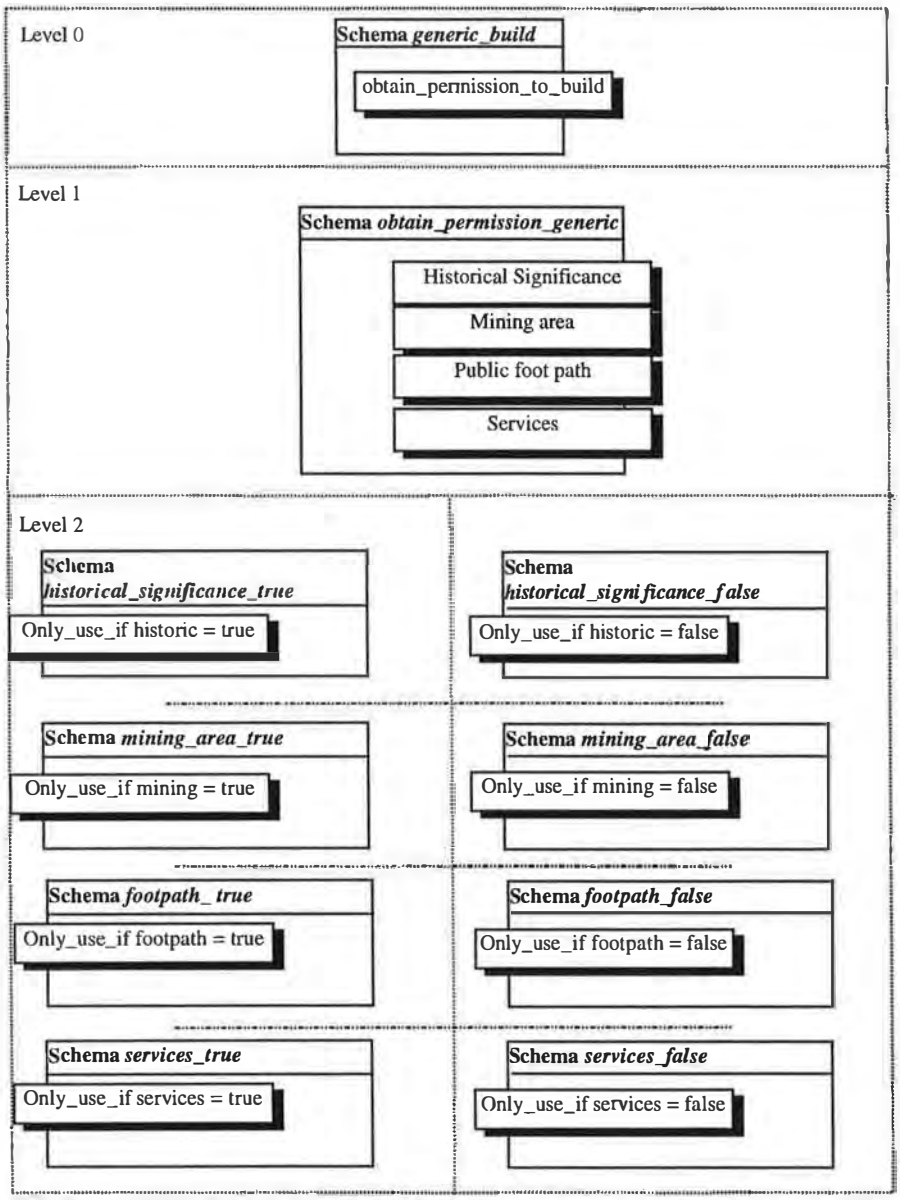


Figure 4-25, Encoding of the multiple conditions case

If the domain regulations are modified to make the actions and ordering constraints required to obtain permission to build different for each combination of values the conditions may take, a method would be required for each combination. As identified in case one, when the number of methods specified increases, so does the probability of redundancy between method specifications.

The observations above concur with the conclusions of case one. Multiple conditions increase the number of method specifications which may potentially be required from equal to the number of values a condition may take in the single case, to the product of the number of values each of the conditions may take in the multiple case.

Case 3: Expressiveness of filter conditions

Encoding task *decorate* identified filter conditions as the mechanism provided by task refinement formalisms for specifying the applicability of different methods for achieving a task. This case considers the expressiveness of filter conditions.

A task's filter conditions are conjunctive i.e. all filter conditions must hold for a task to be applicable. Consider the following generic knowledge structure containing the logical OR connective.

```
to achieve task t
  if condition = value1 or condition = value2 then
      refine using method-a
  else if condition = value3 then
      refine using method-b
  end if
```

The *method-a* is applicable if *condition* is equal to $value_1$ or $value_2$. This knowledge may be encoded by splitting each side of the *OR* connective into different tasks. As demonstrated in the figure below, if during refinement *condition* is equal to $value_1$, then schema *filter1* will be selected and then *method-a*. If *condition* is equal to $value_2$ then schema *filter2* will be selected and then *method-a*.

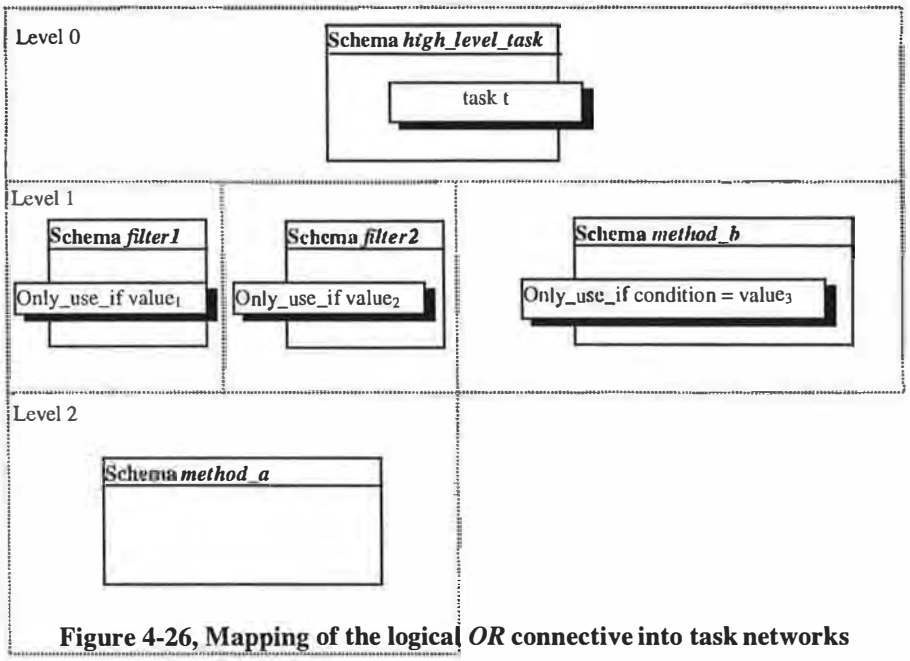


Figure 4-26, Mapping of the logical *OR* connective into task networks

The logical *OR* connective may therefore be achieved with a combination of filter conditions and the schema selection process.

Consider the following nested *if - then - else* knowledge.

```
1. if (condion1 = value1 and condtion2=value3) or (condtion1 = value5) then
2.    if condtion5 = value6 and condition4 = value2 then
3.       refine using method-a
4.    else
5.       refine using method-b
```

Figure 4-27 presents a initial attempt at encoding of this knowledge using a combination of the task selection process and filter conditions. The knowledge above specifies the conditions which must hold for a task *T* to be refined to *method-a* or *method-b*. Line one of the conditions is mapped to the two schemas at modelling level one. Both *filter1* and *filter2* contain the same non-primitive task *T2*. Both schemas *filter3* and *method-b* are possible refinements of *T2*. Schema *filter3* will be selected only if its filter conditions hold. If *filter3* is selected *method-a* will be used to refine task *T2* and therefore *T*.

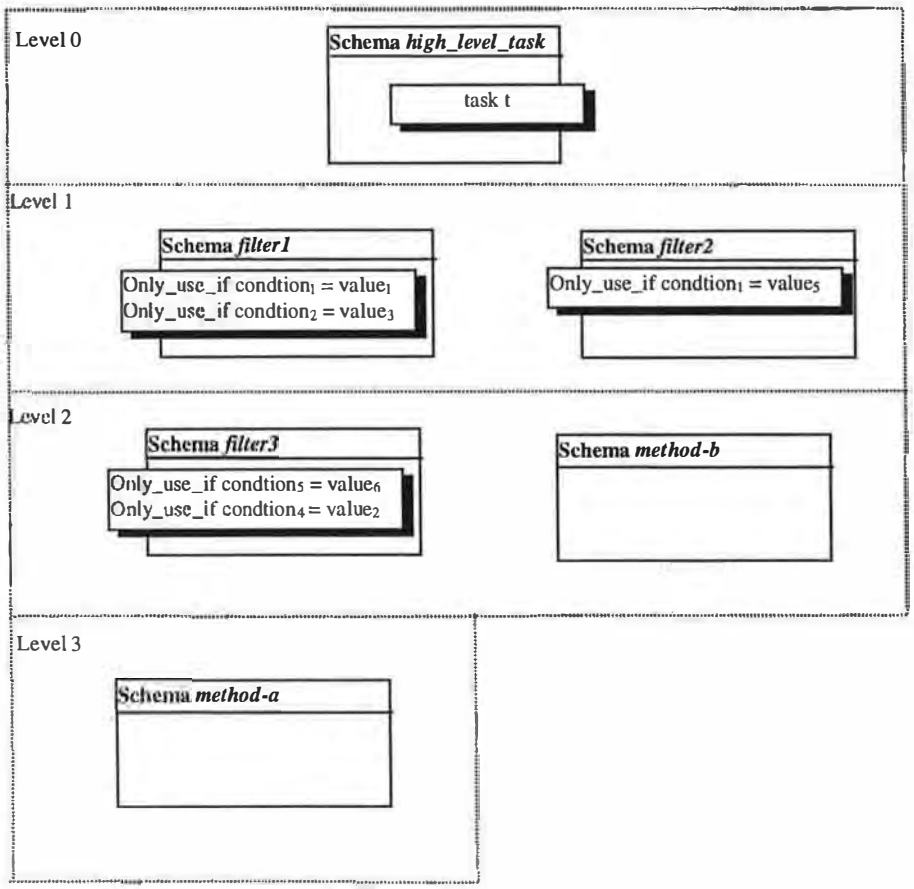


Figure 4-27, Nested if - then - else encoding in a task refinement formalism

This first attempt at encoding does not match the requirements of the domain knowledge. The knowledge specifies that *method-b* should be used if and only if *condtion5* does not equal *value6* and *condtion4* does not equal *value2*. When the task refinement planning engine seeks an expansion to *T2* at modelling level one, both schemas *filter3* and *method-b* will be considered. *Filter3* will be selected if and only if its filter conditions hold. If the filter conditions of schema *filter3* do hold there can be no guarantee the planner will not select *method-b*. Task refinement planners (in the general case) non-deterministically select a method from the applicable set available. To correctly achieve the semantics of the *if-then-else* construct, a filter condition must be added to *method-b* stating that it may only be used when the filter conditions of schema *filter3* does not hold. The modified *method-b* is depicted in Figure 4-28.

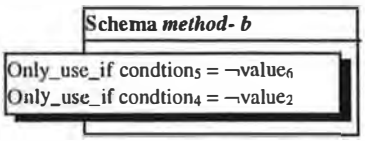


Figure 4-28, Modification to *method-b* to complete the *if-then-else* encoding

The need for additional filter conditions to achieve the *else* part of an *if-then-else* statement adds to the maintenance overhead of a domain's encoding. If the *if* clause requires modification, the *else* clause's filter conditions will also require modification.

In conclusion, filter conditions and the task refinement process may be combined to capture conditions in the form nested *if-then-else* structures for specifying methods applicability. However, the syntax of the encoding is distant from the original *if-then-else* knowledge structure.

4.5.1.4 Encoding task *decorate*

Assume that the decorate task must take into account the number of rooms in a specific design and the final wall coverings selected for each room. The variable number of rooms issue may be addressed using a combination of the type *ROOM* and instances of this type may be used to specify the actual number of rooms in a specific design. An example encoding is depicted below:

```
{front-room, home-office} type ROOM;
```

To specify the final wall covering of a room, a condition may be written in a planner's *always context* (i.e. facts which do not change during the planning process). An example specification is depicted below.

```
front-room final-surface-covering = wallpaper,
home-office final-surface-covering = paint;
```

A decorate schema must account for both the variable number of rooms within a design and the variations in wall coverings. The Task Formalism provides the constructs *foreach* and *iterate* for addressing this type of requirement. An example of each (taken from the Task Formalism manual from an example transporting aircraft to grid locations) is depicted below.

```
N iterate action {fly_to ?way_point}
        for ?way_point over ({100 50} {200 60} {150 40})

N foreach action {counter_problem ?problem}
        for ?problem over (issue_1 issue_2)
```

Figure 4-29, Example *foreach* and *iterate* constructs (from (Tate, Drabble, & Dalton 1994b, pp51))

The *iterate* construct will generate a set of totally ordered actions of the form *fly_to ?way_point* for each member of the set specified by the *over* element. In the example above, the construct will generate the following schema.

```
Nodes   N   action   {fly_to {100 50},
  X   action   {fly_to {200 60},
  Y   action   {fly_to {150, 40};
orderings
  N -->X, X-->Y;
```

The *foreach* construct differs from the *iterate* construct only by the ordering constraints placed on the actions generated. *foreach* permits all the actions in the schema generated to be executed in parallel. *Iterate* adds sequential constraints between the actions generated.

Using the *foreach* and *iterate* constructs in combination with the *final-surface-covering* condition permits schemas of the type depicted below to be written. The schema will generate a paint action for each room which is a member of the set *final-surface-covering* = *paint* and a wallpaper action for each room which is a member of the set *final-surface-covering* = *wallpaper*.

```
schema decorate;
  N foreach action {paint ?room}
      for ?room over (set of rooms  defined as type painted);

  N foreach action {wallpaper ?room}
      for ?room over (set of rooms defined as type wallpaper};

  only_use_for_effects
      rooms decorated = true;
end schema
```

In the specific house building example above, this schema will be instantiated as follows. The encoding assumes the methods detailing the actions for painting and wallpapering a room are specified at a lower modelling level.

```
schema decorate_specific_house;
  nodes   1   {paint home-study},
        2   {wallpaper front-room};

  only_use_for_effects
      rooms decorated = true.
end_schema;
```

This encoding method is successful when the actions within a domain do not interact. To demonstrate this point, consider the following addition to the decoration specification. Electrical work (i.e. wiring and power points) may be hidden below the final wall covering or be placed above the final wall covering. The requirements of each room may be formalised as follows:

```
front-room electrical-work = hidden
home-study electrical-work = exposed
```

The domain knowledge governing the relationships between final wall covering and rough electrical work is specified in the following text.

1. If the electrical work is hidden, then the rough electrical work must be installed before the final wall covering is applied.
2. If the electrical work is exposed, then the final wall covering must be applied before the rough electrical work is installed.

The Task Formalism representation of this situation is depicted below. Two schemas are written for both the paint and electrical work tasks. One schema for each value *electrical-work* may take. In the *hidden* case, an unsupervised condition is added to the *paint* task indicating that the electrical work should be ordered before the *paint* task. In the *exposed* case, an unsupervised condition is added to the electrical work task indicating that the paint task should be completed before the electrical task.

```
schema paint1;
  expands {paint ?room};
  nodes
      1    primitive {paint ?room};
  conditions
      only_use_if  ?room electrical work = hidden,
      unsupervised ?room electrical-work completed at 1;
end_schema;

schema paint2;
  expands {paint ?room};
  nodes
      1    primitive {paint ?room};
  conditions
      only_use_if  ?room electrical-work = exposed;
end_schema;

schema electrical1;
  expands {wire up ?room};
  conditions
      only_use_if ?room electrical-work = exposed,
      unsupervised painting ?room completed at 1;
end_schema;

schema electrical2;
  expands {wire up ?room};
  only_use_if  ?room electrical-work = hidden;
end_schema;
```

This encoding raises the following issues. First, knowledge describing the relationship between decoration and electrical work is distributed between four schemas. Second, ordering constraints are specified through the *unsupervised* condition type; the condition types move responsibility for identifying the action that supports a condition from the domain writer to the planning system's engine.

Distributing the knowledge about the relationship between decoration and electrical work makes domain writing and maintenance more error prone. The domain writer must consider and potentially edit four schemas if the ordering relationship changes. The use of an *unsupervised* condition type places the responsibility on the planning engine for identifying the action which achieves the condition it prefixes and adding ordering constraints to a plan to establish the constraint. If the relationship between them could be specified within the task *decorate,* a supervised condition may be used, hence, informing the planning of the producing action for the condition.

Both the issues identified above result from the encapsulation unit of the schema. Consider the schema *decorate_extended* below. The bolded construct at the bottom of the schema is not supported by task refinement formalisms but demonstrates the encapsulation argument. The semantics of the construct are that after the *foreach* constructs have generated the appropriate *paint* and *wallpaper* actions, the bolded condition recurses over each of these actions and adds ordering constraints dependant upon the state of the *wiring* condition for a room.

```
schema decorate_extended;
  N foreach action {paint ?room}
        for ?room over (set of rooms defined as type painted);

  N foreach action {wallpaper ?room}
        for ?room over (set of rooms defined as type wallpaper};

  only_use_for_effects
        rooms decorated = true;
  conditions
        for all {wallpaper ?room} and {paint ?room} actions
              if {wiring ?room} = exposed then order {decorate ?room} before {paint room}
or                                          {wallpaper ?room},
            if {wiring ?room} = hidden then order {install wiring ?room} before {paint
                                          ?room} or {wallpaper ?room};

end-schema
```

Specifying the ordering knowledge at this modelling level has two advantages. First, the knowledge is specified in a single a schema. Thus, both the processes of understanding and maintaining a domain description are simplified. Second, the *supervised* condition type may be used as all actions affected by the knowledge are described within a single schema. Hence, the planning engine does not need to work to establish the conditions.

In conclusion, the issues identified in section 4.5.1.3 *Encoding the task obtain permission to build* are confirmed when encoding the task *decorate.* The decorate task adds the complexity of a variable number of actions dependent upon the number of objects within a specific domain problem. The *foreach* and *iterate* constructs support a variable number of actions, but there are no constructs to introduce variations in conditions and effects within a schema. To utilise all the constructs supported by task refinement planning, a domain writer must provide a schema for each combination of entities within a domain. In the decorate case this would equate to one schema for the case of one painted and one wallpapered room, one schema for the case of two painted and one wallpapered rooms etc.

4.5.2 Pacifica

This section first identifies the domain knowledge underlying the Pacifica domain (Reece et. al. 1993). The definition identified is then enhanced to include facets of the problem not currently represented, and limitations with the representational devices supported by classical planners are identified.

Appendix C summarises the Pacifica domain.

4.5.2.1 Identifying the domain knowledge in the Pacifica problem

The Pacifica task *Operation Columbus* is divided into three sub tasks: the deployment of evacuation equipment, the utilisation of that equipment to evacuate an island to a central point, and the return of the evacuation equipment and evacuees to a safe location. The knowledge behind the encoding of each task is identified in turn below.

Phase one of Operation Columbus deploys a number of ground transports (GT's) and air transports (AT's) from a military base in a friendly location to the location which is to be evacuated. Two schemas *transport_ground_transports* and *transport_helicopters* (specified in Appendix C) are provided to describe the methods for achieving these two elements of Operation Columbus. Each schema records knowledge about the number of transport types which are to be located and the cargo craft which will effect their deployment.

Phase two of Operation Columbus utilises the ground and air transports positioned in phase one to evacuate a number of cities to a central point. Two schemas *road transport* and *air transport* are provided to describe the methods for achieving the evacuation of a city via the two transport methods available. Both schemas contain knowledge about the steps required to physically move a transport vehicle to a location, load that vehicle, and return it to the central evacuation point. The schemas detail how many people may be evacuated by each trip of a transport vehicle and the calculation of the number of people remaining at the evacuation location after the transport vehicle's capacity has been reached.

Phase three of Operation Columbus loads the evacuation equipment and evacuees onto appropriate transport aircraft and returns both equipment and people to a safe location. The schemas *transport ground transports* and *transport helicopters* (specified in appendix C) are used again in addition to the *fly passengers* schemas. The *fly passengers* schema encodes knowledge about the loading and flying of a transport air craft

With the knowledge behind the Pacifica domain identified, the following sections consider the expansion of each stage of Operation Columbus and assess the Task Formalisms ability to represent and reason which such modifications.

4.5.2.2 Phase 1 - locating evacuation equipment

Figure 4-30, below, presents the encoding of *transport_ground_transports* produced by (Reece et. al. 1993). The current representation contains two limiting issues.

First, the **bolded** lines highlight the encoding of the number of ground transports and air transports available to a mission. The current encoding is static, i.e. the schemas explicitly encode the ground and air transports within a mission and the cargo aircraft which carry them to the evacuation site. If the number of air or ground transports available to a specific mission was to vary, both schemas would require modification. Modification would also be required if the relationship between the air and ground transports and the cargo aircraft was to change.

Second, the schemas contain similar actions, conditions, and effects as the process of transporting air transports and ground transports is similar. Specifically, lines *6 to 11* and *39 to 44* duplicate the same load, take off, fly to, and land actions. Lines *16 to 19, 26 to 29* and *48 to 51, 53 to 59*. duplicate the conditions required by the take off and land actions from a runway. Lines *13 to 19* and *46 to 50* duplicate the conditions on the location of cargo equipment and the cargo equipment's cargo before loading may commence. If knowledge affecting any of these areas was to change, both schemas would require modification.

The current representation of the task of locating evacuation equipment is both inflexible and contains redundancy between schemas. This section examines the ability of task refinement representations to address these issues.

Assume that the number of ground transport and air transports available varies between missions. For example, one mission may have two ground transports and four air transports available, whilst another ten ground transports and zero air transports. The number of cargo transports available to a specific mission may also vary. An example specification is depicted below.

```
GT1, GT2, GT3: GROUND-TRANSPORTS
AT1, AT2, AT3, AT4: AIR-TRANSPORTS
C140, C150: CARGO-TRANSPORTS
```

Assume that the relationship between transport vehicles and cargo aircraft is specified as follows:

```
C140 carries GT1, GT2, GT3
C150 carries AT1, AT2, AT3, AT4
```

Note *GT* = Ground Transport and *AT* = Air Transport

```
1. schema transport_ground_transports;
2. expands {transport_ground_transports ?from ?to};
3. vars      ?from = ?{type air_base},
4.           ?to  = ?{type air_base};
5. nodes
6.     1 action {load ground_transports},
7.     2 action {take_off_from ?from},
8.     3 action {fly_to ?to},
9.     4 action{land_at ?to},
10.    5 action {unload ground_transports};
11.orderings 1->2, 2->3,3->4,4-5;
12.conditions
13.    achieve {at c5} = ?from at 1,
14.    unsupervised {location_gt GT1} = ?from at 1,
15.    unsupervised {location_gt GT2} = ?from at 1,
16.    unsupervised {runway_status_at ?from} = clear at begin_of 2,
17.    supervised {runway_status_at ?from} = inuse at end_of 2 from begin of 2,
18.    unsupervised {runway_status at ?to} = clear at begin_of 4,
19.    supervised {runway_status_at ?to} = in_use at end_of 4 from begin of 4;
20.effects
21.    {at c5} = ?to at 5,
22.    {location_gt GT1} = ?to at 5,
23.    {location_gt GT2} = ?to at 5,
24.    {in_use_for GT1} = available at 5,
25.    {in_use_for GT2} = available at 5,
26.    {runway_status_at ?from} = in_use at begin_of 2,
27.    {runway_status at ?from} = clear at end of 2,
28.    {runway_ststus at ?to} = in_use at begin of 4,
29.    {runway_status at ?from} = clear at end_of 4;
30.end_schema;
31.
32.Schema transport_helicopters;
33.
34.expands {transport_helicopters ?From ?to};
35.
36.vars    ?from    = ?{type air_base},
37.        ?to      = ?{type air_base};
38.nodes
39.    1 action {load air_transports},
40.    2 action {take_off_from ?from},
41.    3 action {fly_to ?to},
42.    4 action{land_at ?to},
43.    5 action {unload air_transports};
44.orderings 1->2, 2->3,3->4,4-5;
45.conditions
46.    achieve {at c141} = ?from at 1,
47.    unsupervised {location_at AT1} = ?from at 1,
48.    unsupervised {runway_status_at ?from} = clear at begin_of 2,
49.    supervised {runway_status_at ?from} = inuse at end_of 2 from begin of 2,
50.    unsupervised {runway_status at ?to} = clear at begin_of 4,
51.    supervised {runway_status_at ?to} = in_use at end_of 4 from begin of 4;
52.effects
53.    {at c140} = ?to at 5,
54.    {location_gt AT1} = ?to at 5,
55.    {in_use_for AT1} = available at 5,
56.    {runway_status_at ?from} = in_use at begin_of 2,
57.    {runway_status at ?from} = clear at end of 2,
58.    {runway_ststus at ?to} = in_use at begin of 4,
59.    {runway_status at ?from} = clear at end_of 4;
60.end_schema;
```

Figure 4-30, Original encoding of schema *transport_ground_transports* and *transport_helicopters*

The first stage of a domain description capable of varying the number of cargo aircraft, the number of air and ground transports, and the relationship between cargo and cargo aircraft is depicted below. Schema *load_and_locate_cargo_for_mission* specifies the task of achieving the first phase of *Operation Columbus.*

```
schema load_and_locate_cargo_for_mission;
  expands {load_and_locate_cargo_for_mission ?from ?to};
  nodes
      N for each action {load_and_locate ?transport ?from ?to}
      for ?transport over {set of cargo transports};
end schema;
```

The schema loops over each instance of the type *CARGO_TRANSPORT*, generating a *load_and_locate* action. The schema below details the results of *load_and_locate_cargo_for_mission* if applied to the sample initial domain description above with a *?from* location of *UK* and *?to* location of *Pacifica.*

```
schema load_and_locate_cargo_for_mission ;;instantiated
  expands {load_and_locate_cargo_for_mission UK Pacifica};
  nodes    1 action {load_and_locate c140 UK Pacifica},
           2 action {load_and_locate c130 UK  Pacifica};
  end schema;
```

Schema *load_and_locate_cargo_for_mission* generates a *load_and_locate* task for each of mission's transports. *Load_and*_locate has the responsibility of loading cargo onto a cargo craft and transporting the cargo craft from the *?from* location an to the *?to* location. The expansion for this task, schema *load_and_locate,* is depicted below.

```
schema load_and_locate;
  expands {load_and_locate ?cargo ?from ?to};
  nodes    1. action {load ?cargo},
       2. action {takeoff ?cargo ?from},
       3. action {fly_to ?cargo ?to},
       4. action {land_at ?cargo ?to},
       5. action {unload ?cargo};
  orderings 1-->2, 2-->3, 3-->4, 4-->5;
  conditions
       achieve {at ?cargo} = ?from at 1,
       unsupervised {runway_status_at ?from} = clear at begin_of 2,
       supervised {runway_status_at ?from} = inuse at end_of 2 from begin of 2,
       unsupervised {runway_status at ?to} = clear at begin_of 4,
       supervised {runway._status_at ?to} = in_use at end_of 4 from begin of 4;
end schema;
```

The *take off, fly to,* and *land at* actions are expanded as in previous encoding. Only *load* task is different, and therefore specified below.

```
schema load_cargo_onto_transport;
  expands {load ?cargo};
  nodes    N for each action {load_onto ?cargo ?transport}
           for ?cargo over {set of transports defined as carried by ?transport};
end schema;
```

The schema generates a *load_onto* action for each transport (air or ground) associated with a cargo aircraft through a *carried-by* relationship. The two instantiations of this schema which will be generated are depicted below.

```
schema load_cargo_onto_transport ;; instantiated;
  expands {load ?cargo};
  nodes    1 action {load GT1 C140},
           2 action {load GT2 C140},
           3 action {load GT3 C140};
end schema;

schema load_cargo_onto_transport ;; instantiated;
  expands {load ?cargo};
  nodes    1 action {load AT1 C130},
           2 action {load AT2 C130};
end schema;
```

This encoding is presented graphically in Figure 4-31 below. Level zero corresponds to the mission task definition. The first phase of level zero refines to the level one schema *load_and_locate_cargo_for_mission.* Level one generates a *load_and_locate* task for each cargo aircraft within a domain (indicated by the * notation used within the figure). *load_and_locate* contains the actions to load the cargo onto a cargo transport and the to physically move the cargo transport from its initial location to the evacuation location. The *load* sub task generates a load action for each of the cargo items to be loaded onto a cargo craft.

The encoding below works because there is no interaction between the *load_and_locate* tasks of for each cargo transport and the loading of cargo onto cargo transports. All the ordering constraints required within the domain may be specified at modelling level two

In conclusion, the Pacifica domain demonstrates the effectiveness of task refinement planning's representational devices within domains which may be formulated into methods which do not interact.

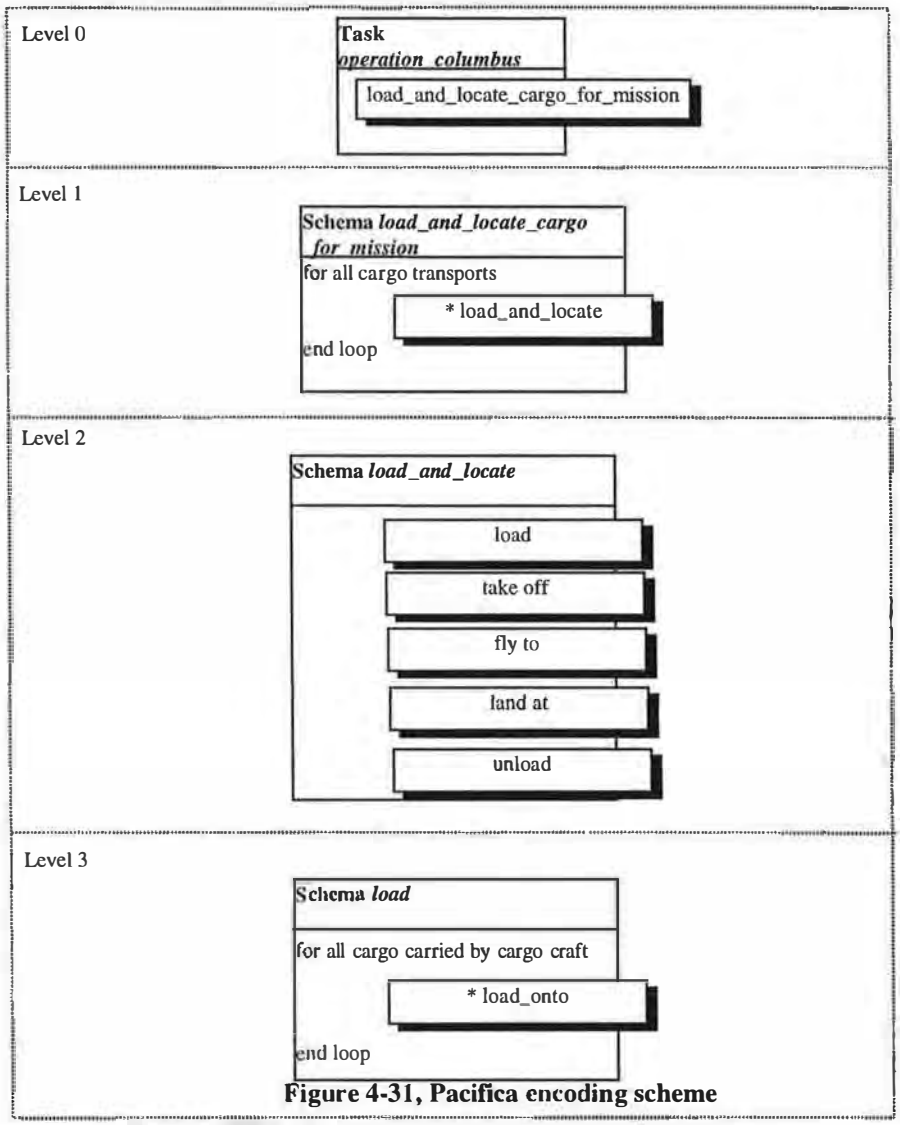


Figure 4-31, Pacifica encoding scheme

4.5.2.3 Phase 2 - evacuating the cities

The current representation utilises precondition achievement behaviour to search the possible combinations of transport types against the number of evacuees in each city to be evacuated. The representation may therefore adjust itself to a variable number of transports and evacuees.

4.5.2.4 Phase 3 - returning evacuees and equipment

The issues surrounding the returning of evacuation equipment and evacuees are identical to those identified in section 4.5.2.2 Phase 1 - locating evacuation equipment.

4.6 Evaluating model-based planners

This section examines the encoding of the flight simulator construction domain in a model-based planner (MBP) formalism. The domain definition is enhanced and the MBP's ability to represent and reason with the enhancements is examined. The flight simulator construction domain is summarised within Appendix C.

4.6.1 Flight simulator construction

Figure 4-32 presents a fragment of the MBP representation of the flight simulator domain (Marshall 1988). The figure is a class diagram which specifies the structure a specific flight simulator instance will follow. The *FLIGHT-SIMULATOR* class may be decomposed via the *sub* relation into a number of instances of class *COMPONENT*. Each instance of class *COMPONENT* will be further decomposed through the *sub* relations into a instance of class *DOCUMENTATION*. An instance of class *DOCUMENTATION* may have three actions associated through the *i.action* relationship. The relation *i.action* indicates that inference is required to determine if each action in the set should be associated with an instance of class *COMPONENT's* documentation. The specification of this assessment knowledge will be discussed latter.

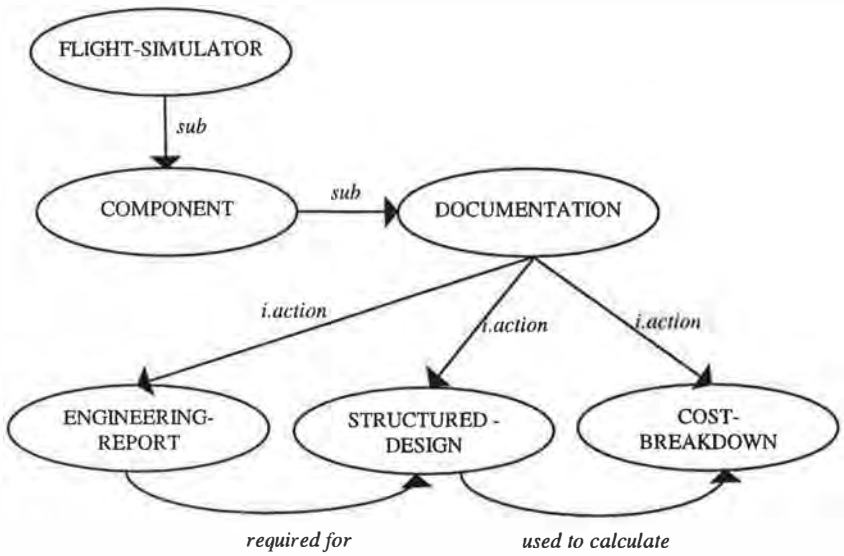


Figure 4-32, Fragment of the MBP representation of the flight simulator domain

A specific instance of a flight simulator is depicted below.

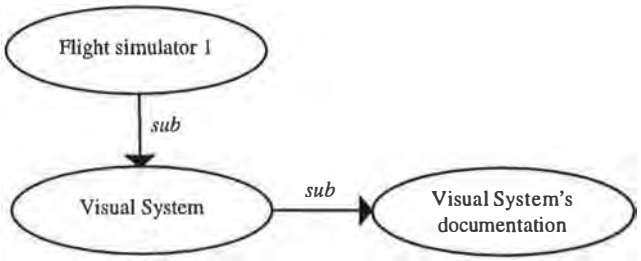


Figure 4-33, A flight simulator instance

During the planning process, the MBP algorithm will recurse through this model assessing the actions associated with each component through the *i.action* relationship. Within Figure 4-33, only the instances of class *DOCUMENTATION*, Visual System's document, have potential actions. In the case of class *DOCUMENTATION,* the domain writer provides production rules for deterring if an instance of the classes *ENGINEERING-REPORT, STRUCTURED-DESIGN*, or *COST-BREAKDOWN* are required for a specific component. A fragment of this rule set is depicted below:

```
rule-base visual system documentation
 if the-visual-system is bought-in AND visual-system.supplier NOT BAA approved then
   require ENGINEERING-REPORT
   require STRUCTURED-DESIGN
   require COST-BREAKDOWN
end if
if the-visual-system is stock-item AND BAA-approved-component then
   require COST-BREAKDOWN
end if
```

The MBP invokes this rule-set to determine which actions should be associated with a specific component. Figure 4-34 below depicts the flight simulator instance with action assessment completed.

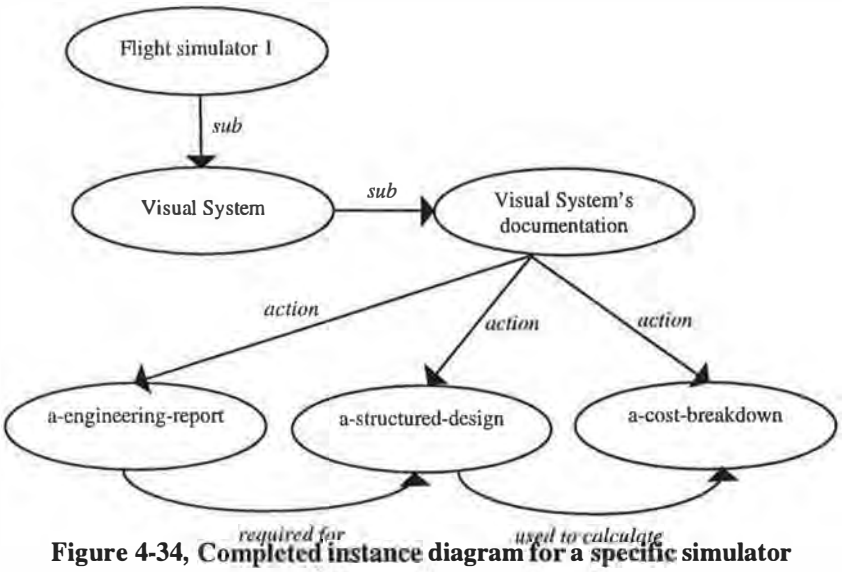


Figure 4-34, Completed instance diagram for a specific simulator

A similar production rule mechanism may be attached to the relationships between actions. For example, a structured design may only require an engineering report if the component to which it is attached is a non standard component. Figure 4-34 includes the relationships between actions generated by this knowledge.

Dependency constraints may be synthesised within MBP between components which are explicitly related. The technology does not incorporate condition and effect reasoning. Hence, it is not possible to specify that an action requires *management approval for bidding = given* without explicitly stating the action which will provide that condition. Without condition and effect reasoning, MBP cannot establish causal structures and protect them for interactions between actions.

In summary, MBP supports rule-based reasoning for determining the action and ordering constraints which should be associated with a specific product. The absence of condition and effect reasoning within the technology, however, prohibits the detection of action interactions and the specification of an action's preconditions without explicitly stating the action or actions which will achieve them.

4.7 Summary and conclusion

When restricted to task refinement, HTN planning's representational devices encode a domain into a number of partial plan fragments or methods for achieving tasks. Several methods may be encoded for achieving each task, and the applicability of each method determined through the filter condition construct. Encoding the *decorate* and *obtain_permission_to_build* extensions to Tate's House Building domain identified a number of issues with this method - filter condition mechanism:

- As the number of conditions affecting the selection of a method increases, so does the number of methods that must be specified for achieving a task.
- In the case of a single condition affecting the selection of a method, in the worst case, the domain writer must provide a method for each value the condition may take.
- If a number of conditions affect the selection of a method, in the worst case, the domain writer must provide a method for each combination of values the conditions may take.
- Separating the actions dependant upon a condition form those actions independent within a method into different task networks is effective providing the ordering constraints between the actions are not also dependant upon the conditions. Introducing new modelling levels prohibits the use of supervised condition type and ordering constraints due the encapsulation constraints upon task networks.

Encoding the *decorate* task highlighted a specific problem which occurs when using the *foreach* and *iterate* constructs in domain descriptions with a variable number of entities. The absence of conditional ordering and condition constructs prohibits the inclusion of ordering constraints and conditions with each action generated by the *foreach* and *iterate* constructs. Hence, the domain writer is forced to write these constraints at lower modelling levels with the associated encapsulation constraints on the constructs that may be used. This issue is particularly important in the case of a domain with a variable number of entities, as writing methods for all possible combinations of entities is unfeasible.

The analysis of the expressiveness of filter conditions lead to the following conclusions.

- It is possible to use filter conditions in collaboration with the task refinement process to achieve the logical *OR* connective and nested *if-then-else* structures.
- The syntax of such an encoding is distant from the construct the domain writer wishes to encode. Thus, increasing the cognitive overhead of the encoding process and the understanding a domain description.

- The *else* clause of a *if-then-else* must have the negation of the *if* statements condition specified as task selection is non-deterministic. This redundant specification increases the maintenance overhead of a domain's description.

Encoding the Pacifica domain demonstrated the effectiveness of task refinement planning in domains where the tasks may be encoded as relatively interaction free units.

The limitations identified above may be addressed if the task refinement technique is used in conjunction with the precondition achievement functionality supported by task refinement planners. However, the use of precondition achievement in the examples from which the limitations are derived would require a planner to search for actions and ordering constraints for which domain knowledge is available to determine. It is the ability of task refinement planning to represent this knowledge which is being criticised.

Turning to MBP, the flight simulator encoding highlighted the expressive power of MBP formalisms at capturing domain knowledge that maps into production rules. However, the absence of condition and effect reasoning enforces ordering constraints to be specified only if the producing and consuming action are known. It is not possible to specify action preconditions, leaving their establishment to the MBP.

This analysis leads to the following conclusions. First, the limitations of task refinement formalisms are justified within the context of the computational complexity of partially ordered and instantiated plans. In such a context, determining the truth of a statement is computationally expensive. Thus, task refinement developers have limited the expressiveness of the constructs their formalisms support to prevent domain writers specifying computationally intractable domain descriptions. However, all the conditions upon which the cases above depend upon do not change during the planning process and therefore are not subject to the computational complexities of determining truth. Hence, more expressive formalisms may be used.

MBP planning operates only in the space of domain facts that do not change during the planning process. Hence, the technology has developed a highly expressive formalism that maps closely to application domain knowledge. MBP contains no mechanisms for establishing and maintaining causal structures in a plan.

The rationale for integrating classical and model-based technologies developed in Chapter 2 may be refined as a result of these conclusions. Task refinement planning is designed to address the computational complexities inherent when determining truth over a partial-order plan. MBP is designed to exploit domain

experts' knowledge within the computationally inexpensive space of facts outside of the evolving world state within a partial-order plan.

An integrated architecture would permit the expressive MBP formalisms to be deployed for generating the actions and ordering constraints obtainable from an expert's knowledge of a domain. Classical planning techniques may then be used to establish and maintain causal structures.

This type of integration is supported by Nau, Gupta and Regli (1995).

> *Since AI planning researchers are usually more interested in general conceptual problems than domain-dependent details, the AI approach to manufacturing planning has typically been to create an abstract problem representation that omits unimportant details, and look for some way to solve the abstract problem. From the point of view of manufacturing engineers, these "unimportant details" often are very important parts of the problem to be solved.... Manufacturing planning researchers typically want to solve a particular manufacturing problem and present their research ... without discussing how the approach might generalise to other planning domains.*
>
> (Nau, Gupta and Regli 1995)

The analysis presented in this chapter is based upon the author's generation of counter examples from existing domain representations. To confirm and identify other issues encountered in application domains, Chapter 5 elicits the planning knowledge utilised by experts in the construction industry. Chapter 6 then considers the task of encoding this knowledge within task refinement and model-based formalisms.

5. Elicitation of planning knowledge from the construction industry

An expert is one who knows more and more about
less and less.
Nicholas Murray Butler (1862–1947)

5.1 Introduction

Chapter 4 identified the benefits of basing planning research within industrial planning domains. This chapter describes the elicitation of planning knowledge from the construction industry; a domain not previously extensively studied by classical researchers. The resultant knowledge is used within Chapter 6 to verify and extend the limitations identified in Chapter 4 within an industrial context.

Within this chapter, knowledge elicitation is defined as a compound task consisting of knowledge acquisition and knowledge modelling, where knowledge acquisition is the task of identifying domain knowledge from experts, documentation etc. Knowledge modelling is the task of combining the elicited knowledge to produce a model of a domain.

Whilst the primary aim of the research presented in this chapter was the actual elicitation of construction knowledge, the key secondary aim was that this elicitation should be independent of the aims of this thesis. The results will, therefore, provide a generic model of construction planning knowledge which is not skewed towards the overall aims of this thesis.

Knowledge acquisition was effected through a set of interviews, observations of experts planning, and observations of the use of plans on a construction site. Knowledge modelling was achieved through the KADS methodology. This chapter initially discusses the reasoning behind the acquisition and modelling approach and the selection of the collaborator. The discussion then moves to describe and present the resultant KADS models.

5.2 Selecting an application domain and a collaborating organisation

The construction domain has had a limited history within classical planning literature. Publications have centred upon Tate's House Building Domain (Tate 1976) as a demonstration of the capabilities of task refinement planning systems (Kartam, Levitt & Wilkins 1991). Therefore, unlike the military evacuation and logistics domain at the centre of ARPI, construction planning has not been exhaustively investigated. Further investigation may identify new generic concepts that may be applied to other planning domains.

The Llewellyn Group of Companies was selected as an industrial collaborator for two reasons. First, the organisation is an established construction company with experience of a variety of projects. Second, the organisation was prepared to commit the time and resources necessary for the elicitation process.

Profile of the Llewellyn Group of Companies

The Llewellyn Group of Companies provides a comprehensive range of construction services. Recent refurbishment projects range from the eight million-pound modernisation of tower blocks in central London through to the protection of an ageing water intake jetty for Scottish Nuclear Fuels on the Firth of Clyde. Design and build projects range from a leisure centre in Berkshire to a multimillion-pound divisional police headquarters in Surrey. Recent customers include: British Telecom, J Sainsbury, Inland Revenue, Kent County Council, Sussex NHS trust, and The Employment Service.

The group is based in the south of England at Eastbourne, Brighton, Hastings, London, and Milton Keynes employing around 800 people (excluding labour and material sub contractors).

5.3 Selecting the knowledge elicitation approach

The knowledge acquisition bottleneck (Feigenbaum 1980) was identified by early knowledge-based system development projects (Parsaye & Chignell 1988). Overwhelmed by the complexity of domain knowledge and the difficulties encountered when trying to elicit it, early expert system developers identified the need for tools and methods to support knowledge acquisition.

CommonKADS[1] (Schreiber 1992) was conceived in 1983 with the aim of providing a comprehensive methodology for developing expert systems. A succession of further research projects have developed CommonKADS into a mature tool-supported methodology which aims to become the commercial standard within Europe (KADS Consortium 1997).

By following the leading commercial methodology, the two aims of the knowledge acquisition phase may be addressed. First, industrially proven knowledge modelling techniques can be utilised. Second, by following an independent methodology, the resultant knowledge model will not be skewed towards the aims of this research project.

An additional benefit of KADS is that by using a standard notation, the resultant model may be communicated to a large audience. KADS has recently been used to describe the generic planning knowledge encoded in domain independent planning algorithms (Valente 1995; Barros, Valente & Benjamins 1996; Kingston, Shadbolt & Tate 1996, Benjamins, Barros & Valente 1996).

[1]KADS was originally an acronym for the "Knowledge Analysis and Documentation System." This definition was latter modified to "Knowledge Analysis and Design Support." Today KADS it is used as a proper noun.

5.4 Overview of the KADS methodology

This overview is based upon the defining KADS methodology text (Schreiber & Wielinga & Breuker 1993).

KADS decomposes the knowledge acquisition process into three tasks: elicitation, interpretation, and formalisation. Elicitation is the process of identifying knowledge from domain sources (experts, documentation etc.). The interpretation phase moves the elicited knowledge into a conceptual framework. Formalisation moves the knowledge from the conceptual framework into a form suitable for use by a computer program.

Knowledge acquisition techniques are not defined within KADS. The authors of the methodology assume such techniques are well documented and understood elsewhere. The methodology, however, does indicate where the products of various acquisition techniques feed into the model set (Wielinga, Schreiber, & Breuker 1993 pp 42-43). KADS focus is on the interpretation and formalisation activities, basing the methodology on two principles: multiple models and knowledge-level modelling. Multiple models provide a mechanism for addressing complex systems. Each of the models concentrates on specific aspects of a system whilst ignoring others. Collectively, the different models capture all facets of the system under consideration. This approach is common throughout software engineering e.g. (DeMarco, 1982). Knowledge level modelling is motivated by Newell (Newell 1982). The rationale being a desire to model knowledge from the perspective of why a system performs an action, independently from how this functionality will be realised in rules, frames, logic etc.

The principle of multiple models is realised in KADS through the set of interrelated models. Knowledge level modelling is achieved within a specific model in the KADS set; the model of expertise. Each member of the KADS model set is introduced below.

- **Organisational Model** captures the socio-economical environment of a KBS. The model results in a description of the functions, tasks and bottlenecks within the organisation under consideration. The model predicts how the KBS will influence the organisation and the people working in it.

- **Application Model** defines what problem the KBS should solve within the organisation and what the function of the system will be in that organisation. The model captures external constraints, relevant for the development of an application. Examples being the speed and efficiency of such a system, the hardware, and software available.
- **Task Model** specifies how the function of the system, defined in the application model, is achieved through the tasks the system must perform.

- **Model of Co-operation** identifies the human-machine relationships required by the tasks in the task model. The model differentiates the functionality executed by humans from that realised by the machine.

- **Model of Expertise** forms the central activity in KBS construction. The model specifies the problem solving behaviour of a target KBS through extensive categorisation of the knowledge required to generate this behaviour. Knowledge levels are introduced to separate control from domain concepts.

- **Design Model** describes the computational and representational techniques required too realise the artefact specified in the previous models. It is at this point KADS moves from the logical, implementation independent perspective, to the implementation dependent view.

In summary, KADS provides a set of models for capturing the facets of a system relevant to constructing a knowledge-based system. A set of guidelines and assessment criteria facilitate the construction of the model set.

5.5 KADS models of the construction domain

This section describes how the knowledge elicitation process was achieved, detailing how knowledge was acquired, and the construction of the KADS model set. Before the models are described, the subset of the KADS methodology applied to construction problem is described and justified.

5.5.1 Subset of KADS applied to the construction domain

KADS is designed to support the development of a knowledge-based system for a specific application. The aim of this chapter is to identify and model the planning knowledge within the construction domain. Hence, not all the steps and models within KADS were applied. This sub section presents the rationale for including and excluding specific models to achieve the aims of this chapter.

The KADS methodology provides the organisational and application models to aid a domain modeller in understanding an organisation's business process, social factors, and the inputs and outputs of the planning process. Two factors lead to the decision to develop both models. First, understanding the overall construction business was considered an essential precursor to developing the model of construction domain's planning knowledge. The models will provide the structure of the domain and its terminology. Second, the models will provide the application requirements of the planning process, defining its inputs and outputs. This understanding will form the basis for assessing the ability of classical and model-based planning techniques to address industrial applications.

The task model identifies the high level tasks performed during an organisation's existing planning process. The model was constructed to isolate expert planning "functionality", hence, providing a framework to stimulate discussion. For example, the model led to questions of the type "what knowledge is important within task A, but is not used in any other task".

A model of co-operation was not constructed. The inputs and outputs to a planning application in construction planning are identified by the application model. By definition, an automated planning algorithm will require no user involvement during the planning process.

The KADS model of expertise consists of four layers: *strategic, task, inference,* and *domain knowledge.* The *domain knowledge* layer captures the concepts, concept-properties, and relationships used by human experts when performing the task, in this case planning, that is being considered. The *inference* and *task* layers model the inferences and ordering of those inferences used by human experts receptively. The reconfiguration of task structures under particular problem solving domains is modelled by the *strategic* layer.

The *domain knowledge* layer provides the input to an automated planning system; hence, the model is essential to the aims of this thesis. It was not felt appropriate to model the three remaining layers. Although aspects of the task layer were modelled in conjunction with domain knowledge acquisition. The purpose of the elicitation described in the chapter is to provide a vehicle to motivate and evaluate the integration of classical and model-based planning. The further work section within Chapter 9 discusses the important research direction of building a model of human problem solving within the construction industry and the comparing existing planning technologies with this requirement.

5.5.2 Overview of the construction cases studied and the knowledge acquisition approach

This section overviews the two construction cases studied and the knowledge acquisition process applied to them.

5.5.2.1 Case 1: extension to a supermarket to provide a restaurant facility

The first case considered was an extension to an existing supermarket to provide a new restaurant facility. The project was completed at the time elicitation started. Hence, it was possible to fully evaluate the building's design, the plans used in the construction, and photographs of different phases in the construction.

Figure 5-1 depicts the situation before the construction commenced. The supermarket has an unused sub unit backing onto a planting bed. The project's aim was to convert this space into a restaurant facility, exploiting the existing wall and roof structures.

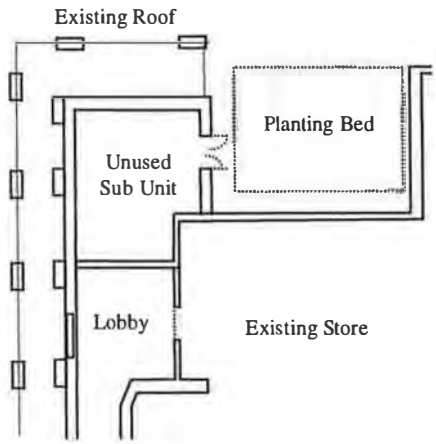


Figure 5-1, Initial supermarket building[2].

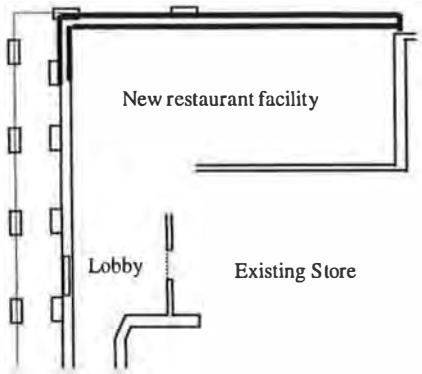


Figure 5-2, Final state of the building

[2] The diagram is not to scale. Approximate area displayed is 300 x 300 m.

Figure 5-2 depicts the final building structure. The original unused sub unit and planting bed form the area of the new restaurant facility. The bolded walls were constructed and the new area decorated.

Knowledge acquisition of the case commenced by studying the set of design diagrams describing the modifications to the building and the plan followed during that construction. The design diagrams detailed the components of the new restaurant facility and their interconnections. The material was studied before the first meeting with the collaborating organisation. Without previous knowledge of the construction domain, it was possible only to relate activities with components. To determine why the activities were required and the reasons leading to the ordering constraints required questioning of domain experts.

The initial contact with domain experts concentrated on the overall construction process through interviews. Questioning covered the process from a customer approaching the organisation to a completed building being delivered to that customer. This elicitation phase lead to the construction and reviewing of the organisation model.

With the organisation model defined, questioning centred upon the planning function within the overall constructing process. The results isolated the planning function to produce the organisational model. Domain experts were used to formal notations (from design notations used in the domain) and readily adapted to understanding and critiquing both the organisational and applications models.

With the context established, elicitation moved to considering the tasks performed during planning. The task model provided a notation to record this knowledge, and as with the organisation and application models, provided a central focus for discussions.

Construction of the domain knowledge layer of the expertise model was structured from the task model. Experts were asked to define the concepts and relationships used in each task. This approach facilitated the gradual modelling of domain knowledge - allowing domain experts to concentrate on each task in turn. Experts at times had difficulty in verbalising their knowledge replying, for example, to questions of the form "why is activity1 ordered before activity2" with "It just has to be done that way." This issue was addressed by phrasing questions as "what effect would moving activity2 before activity1 have." This approach proved successful.

5.5.2.2 Case 2: construction of retirement flats

Elicitation commenced on a project to construct a five-story block of flats as building was about to commence. This case enabled the author to observe a working construction site, offering the opportunity to question the construction workers and managers as planning decisions were being made.

The model set elicited in case one was based upon interviews and hypothetical problem solving exercises. To verify the integrity of the models this second case used weekly visits to an actual construction site. As the design of the flats differed from the design of the supermarket extension (e.g. trench foundations as opposed to piles and beams), the generality of knowledge elicited during the first case study could be tested.

Observing and discussing the use of plans on-site refined the organisation and application models. This was a natural refinement, the expertise of individuals working in the earlier phases of the construction process on latter phases was not as current as those actually working within those phases. The overall structure was found to be correct, just subtle changes in detail were required.

The expertise model was reinforced by actually observing the constraints described during the interviews in case one. For example, one could see the physical size of plant equipment (cranes, mechanical-diggers etc.) and the need to sequence activities around their ability to access different parts of the site. The consequences of omissions in planning detail could also be observed. For example, the damaged caused by plant equipment moving over drainage pipes which would have been avoided had the drains be laid after the plant had completed its work in that area.

5.5.3 Organisational model

The organisational model was elicited through a combination of structured interviews at various levels within the Llewellyn organisation and observations of experts planning. One of the organisation's directors provided strategic organisational knowledge, whilst those involved in the planning of projects supplied detailed planning-specific knowledge. Site managers, the users of the plans, identified how the results of the planning phases are used, problems highlighted, and replanning initiated.

The organisational model is presented in Figure 5-3. The figure shows the main processes, data stores, and external entities in the organisation. The figure is biased to the planning functions within the organisation and omits non-essential detail.

The model encapsulates three different scenarios:

- **Design and Build.** A dialogue between Llewellyn and the customer establishes the requirements for a construction project (the analysis process). From this, a detailed design is produced (the design process). The detailed design is combined with product knowledge to prepare a high level plan (the construction planning process). The plan is used to prepare materials and plant equipment schedules (the material and plant ordering process). Once construction begins, the plan is refined into daily and or weekly plans, depending upon the phase the project has reached. Generally the more trades active on a site, the lower the granularity of planning. More trades result in more interactions between activities; hence, trades must be carefully co-ordinated to ensure efficient working.

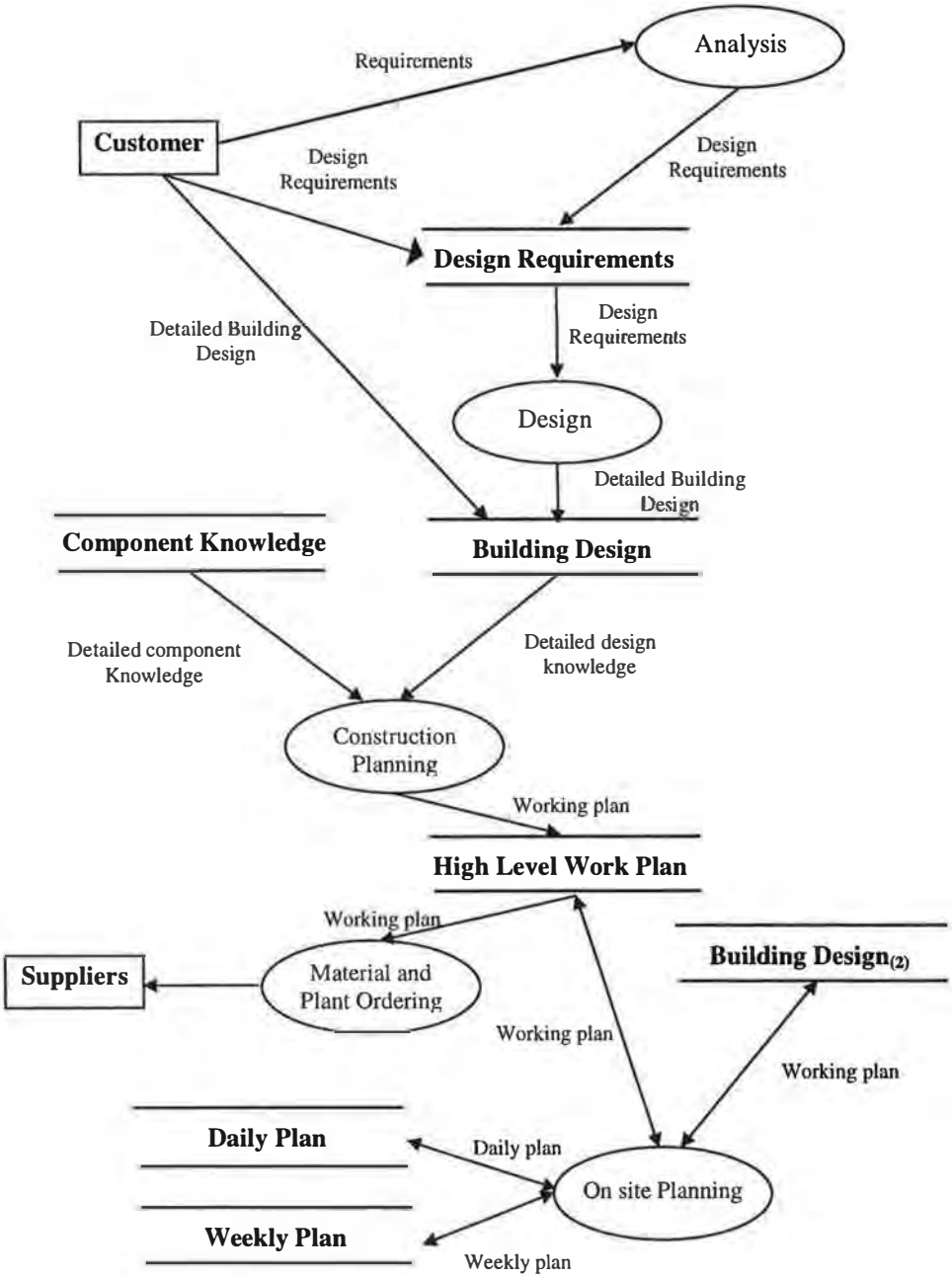


Figure 5-3, Llewellyn organisational model

- **Build.** A customer may provide a detailed building design, requiring the Lewellyn group to construct it. This phase starts at the *construction planning process* and proceeds from that point as in *design and build* case.

- **Bid.** Bidding runs through the *analysis*, *design*, and *construction planning* phases. The resultant building design and construction plan are used to produce cost estimates for realising the project. The customer may or may not accept the bid for reasons of cost or time scales (typically the construction process would be too long). Bidding is therefore a gamble; more time spent designing and planning to produce accurate and competitive estimates, balanced against the risk of losing the bid and the investment in producing it.

The three scenarios presented above represent the typical activities the organisation encounters. There are many different variations; for example a build project may revert to a design and build, if the customer's design proves impractical (the site may be found unsuitable for the foundation construction techniques envisaged after ground tests are completed).

Issues relating planning to the organisations goals are listed below.

1. **Time taken to produce a high level working plan from a buildings design.** Producing the high level construction plan is a slow and costly process, requiring input from highly skilled practitioners from several disciplines. Much emphasis was placed from all involved in the process on the need for experienced people. It is the combination of experience, skill, and collaborative working requirements which make planning a costly and time consuming process.

2. **Knowledge archiving.** The practitioners involved in the planning process are a valuable resource to an organisation. There were indications that a project would not be considered if people with previous experience of the techniques demanded were not available. Concern was expressed at the organisation's strategic level of the effects of key people leaving the organisation. A way of archiving knowledge was desired.

3. **Cost of Planning.** The cost of planning was highlighted in issue 1 but becomes a major issue in bidding. Time spent planning is not guaranteed to be recouped, as the project may not be awarded. However, if planning is not sufficiently detailed, a bid may be won, only to find the cost of construction is greater than the income generated. There is a major desire in the organisation to optimise the planning function to increase its speed and accuracy, while reducing the costs.

4. **Level of detail in plans**. The high level work plan typically considers the building at a high level of abstraction. This is a cost saving practice, as the lower level of detail considered in the planning the phase, the greater the time required to produce a plan. The need for constant on-site planning reflects the limitations of ignoring much of the building's detail at this early stage. Planning to a lower level of abstraction during the construction planning process is desirable.

The points identified above indicate a need for improvements to the planning support available to the construction industry. Socio-economical factors complete the organisational model, and are presented below:

1. **Perception of information technology**. Llewellyn use computer aided design software to support the design process, and office automation technology (spread sheets, databases, word processors) for administration support. Parts of the organisation use no information technology (IT), for example no computers are available on construction sites and a high percentage of design work is carried out on paper. The organisation's perception is therefore polarised from everyday experience of IT, through to little awareness of IT.

2. **Automation concerns.** People involved in both the development and the execution of plans were concerned that automated planning technology would impose constraints upon their working practices. For example, imposing tightly-controlled schedules which may lead to robotic type working conditions. The need for highly configurable tools was stated by all levels within the organisation.

3. **Disbelief.** A number of human planners felt automated planning was simply impossible. The number of constraints considered by humans was felt to be beyond the storage capacity of computer technology. The need to convince people of the feasibility of automated planning before securing their support for the knowledge elicitation process was identified.

4. **Explanation**. A plan with a supporting rationale was considered essential by all levels within the organisation.

5.5.4 Application model

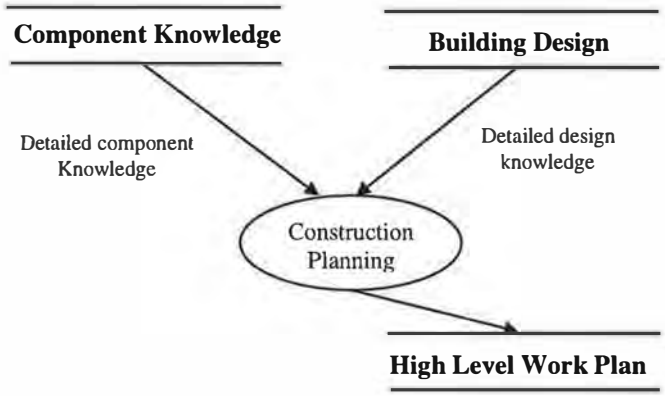


Figure 5-4, Application Model

The application model refines the organisational model (section 5.5.3) by identifying the processes to be addressed by a KBS. The *construction planning* and *on site planning* processes provide the planning functions of the organisation, hence, the processes have been extracted from the organisational data flow (Figure 5-3) together with their input and outputs to produce the application model (Figure 5-4).

The construction planning task takes as its input a building's design and component construction knowledge and produces as its output a plan to realise the design. The resultant *high level work plan* is used to guide the construction. The *on site planning process* refines the *high level work plan* to the level of granularity required by the site manager. Problems with the high level plan may be encountered on site. The site manager modifies the *high level work plan* to reflect this.

The *construction planning* process takes two inputs: component knowledge and a building design. The building design is presented as drawings output from a computer aided design tool. Component knowledge is located in engineers' experience and component manuals from manufacturers.

The *construction planning process* produces one output: a high level working plan. The plan is normally implemented on a presentation and analysis tool (e.g. Microsoft Project™) and can be viewed, for example, as GANTT and PERT charts.

On site planning takes a paper copy of the *high level work plan* in GANTT chart format. The process is carried out using pencil and paper. Daily and weekly plans are produced on paper, utilising the techniques developed by individual site managers. The process may be described as ad hoc, but is highly effective. Site managers develop a mental model of the project they are working on and use model to identify problems.

5.5.5 Task model

The task model refines the process identified in the application model to specify how the function is achieved through a number of tasks.

Figure 5-5 presents the task model derived from observing human planners. Each task is described below:

- **Specify Problem** The human planners spend time examining a building's design to familiarise themselves with it.
- **Identify Activities** Components are viewed at different levels of granularity, and the humans define activities which need to be completed for the building to be constructed.
- **Identify Ordering Constraints** The interrelationships between components are examined to define the ordering relationships between activities.
- **Resource Needs.** By examining the overall set of activities, resources are selected.
- **Review plan.** Several different disciplines examine the plan and identify problems and discuss methods for resolving them.

The process above is initially sequential, before moving to a phase where the human planners move from one task to another as issues arise.

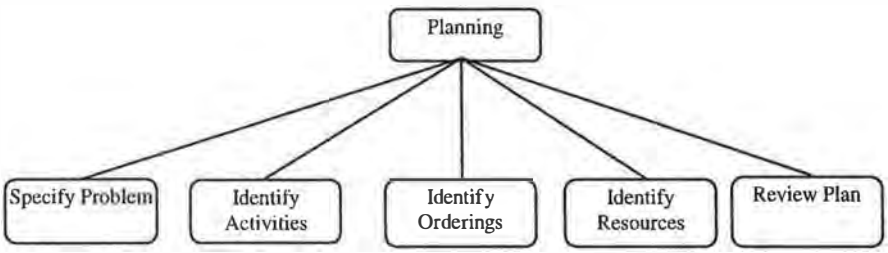


Figure 5-5, Task model of construction planning

5.5.6 Model of expertise

The model of expertise is the most important in the KADS model set; it is the model which differentiates the development of knowledge based systems from traditional information systems. Constructing the previous models developed an understanding of the operation of the company, its terminology, how the planning functions integrate into the business, and the requirements of an automated planning system in this organisation. The goal of the model of expertise is to specify the problem solving expertise required to solve the process identified in the application model: construction planning.

The model of expertise is broken down into four interrelated sub models: domain knowledge, inference knowledge, task knowledge, and strategic knowledge. The separation allows different types of domain expertise to be identified and related.

For the purpose of identifying the domain knowledge utilised in construction planning, only the domain knowledge sub-model is constructed.

5.5.6.1 Domain knowledge sub-model

The domain knowledge model provides the conceptualisation of a domain, for a particular application, in the form of a domain theory. Domain knowledge is partitioned into four categories: concepts, properties, relation between concepts, relation between property expression, and structure. The first three categories are derived in turn below. The structure category is not presented within this chapter, but is presented in Chapter 6. The structure category combines the concepts, properties, and relations between concepts into one diagram.

5.5.6.1.1 Domain concepts

Domain concepts are the central objects in the domain knowledge and may be compared with entities in an Entity Relationship diagram (Chen 1976) and objects in an Object Model (Rumbaugh et. al., 1991). The first task in constructing the domain knowledge sub-model is to identify the domain concepts.

Llewellyn supplied a set of diagrams which together described the components of a building's design. Construction is a mature engineering discipline, hence, its notations are precise and unambiguous. The design documents were complemented with a plan for the construction of the building depicted. The domain experts were asked to describe each diagram and questions were posed to identify the planning knowledge applied. Figure 5-6 and Figure 5-7 present fragments of the diagrams provided. Figure 5-8 depicts a fragment of the construction plan. Table 5-1 presents a transcription of the expert's description of Figure 5-6, Figure 5-7 and Figure 5-8.

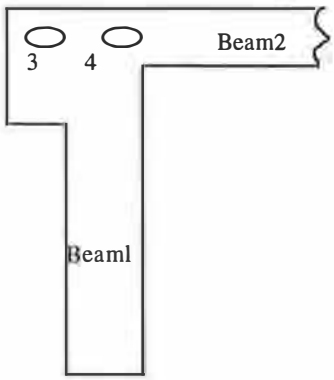


Figure 5-6, Fragment of the Pile and Beam layout diagram

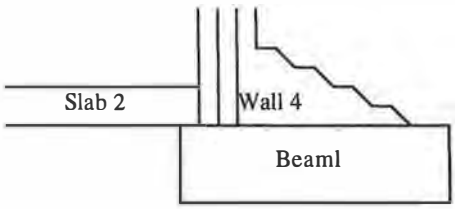


Figure 5-7, Fragment of Beam, Slab and Wall Details Diagram

Figure 5-8, Fragment of the construction plan

Name	Transcription of experts comments
[CHQ.94.2 179.04], (Figure 5-6 above)	The two *ground beams, beam1 and beam2 support* the force of the *walls above* them. *Pile 3* and *Pile4 support* the weight of the *beams. Piles* are *laid* using a *piling machine.* The machine needs a *pile mat* to stop it sinking into the ground. Once the *pile mat* is in place, the pile is *driven* into the ground....
[CHQ.94.2. 176.02], (Figure 5-7 above)	*Beam1* is *supporting* the weight of *wall 4* and *slab2.* The *beam* must be *laid* before the wall and slab. Some of the Wall must be built before the slab, hence removing the need for formwork at the wall end of the slab....
[CHQ.94. 3.178.04] (Figure 5-8 above)	The *pile mat* must be placed before the pile can be driven. Otherwise the *piling rig* will sink into the ground. *Piles* are laid before *beams* because they support the *beams.* The *wall* is started before the *slab* because the slab rests on the beam and against the wall. Hence, working this way removes the need for formwork on the slab, as the wall acts as the formwork.

Table 5-1, Domain experts comments on the design fragments and the plan

Words in Table 5-1 highlighted in *italics* indicate potential concepts or relationships. From the comments on Figure 5-6, the following concepts were identified: *beam1, beam2, beams, pile3, pile4, piling-machine*, and *pile-mat*. Each was recorded as a candidate concept using the KADS domain description language (Schreiber & Wielinga & Breuker 1993, pp 71-91):

concept *beam*	**concept** *beam1*
concept lay-beam	**concept** lay-beam1

The process of reviewing each design diagram with an expert was repeated until the concepts in the specific building were identified.

5.5.6.1.2 Properties

The concepts identified in section 5.5.6.1.1 may have properties associated with them. Properties are defined through their name and a description of the values it may take. Each concept was examined and properties identified. Physical properties (size, weight, etc.) were simple to identify, they are recorded on the building's design drawings. The *beam, beam1, lay-beam, lay-beam1* concepts identified in section 5.5.6.1.1 are refined below to include their properties.

concept *beams*
properties:
length: integer mm
height: integer mm
width: integer mm

concept *beam1*
properties:
length: 5040
height: 1000
width: 2000

concept *lay-beam*
properties:
dependent upon: actions

concept *lay-beam1*
properties:
dependent upon: *Pile3* and *Pile4* being laid.

5.5.6.1.3 Relations between concepts

KADS notes that the most common relations between concepts are the sub-class relation and the part-of relation. These relationships are presented first, followed by the domain specific relationships identified.

Sub-class relationships

Within the modelling phase, identifying the sub-class relationship served two purposes. First, to differentiate between instances and generic knowledge. Second, to identify generic knowledge common between two or more types of concept.

The *beam1* concept identified in the previous section is an instance of the candidate concept *beam*, as *beam1* has all the properties of *beam* but specific values for each property. The concept *beam* and instance *beam1* are presented below. Note the addition of the **instance-of** field in the description of the instance *beam1*. The field records the concept from which the instance is derived. Note once a concept is confirmed as a concept, its name is depicted in uppercase.

concept *BEAM*
    properties:
        length: integer mm
        height: integer mm
        width: integer mm
        stress: compression, tension
instance *beam1*
    instance-of beam
    properties:
        length: 5040
        height: 1000
        width: 2000
        stress: compression

The sub class relationship allowed concepts to be organised into hierarchies, capturing the common knowledge between two or more concept. The example below demonstrates how this principle was applied to piles. The generic *PILE* concept captures the purpose of piles and general domain properties. The more specific *BORE-PILE* and *DRIVEN-PILE* capture knowledge specific to these types. Instances of concept *BORE-PILE* differ from instances of *DRIVEN-PILE* by the method used to place them into the ground. *BORE-PILE* instances are screwed into the ground, where *DRIVEN-PILE* instances are hammered into position. Placing *BORE-PILE* instances results in quantities of soil being deposited on the surface. A plan must therefore include actions to remove this waste. Organising concepts into hierarchies facilitated the discovery of this type of knowledge.

concept *PILES*
knowledge elicitation
description Piles are used in areas where the ground is too soft to support a building's weight directly. By driving a pile into the ground, the force of the building is distributed over a large area.
domain properties Piles need to have their position accurately set out and a pile mat to be laid before they maybe placed.
sub-type-of *OBJECTS*

concept *BORE-PILES*
knowledge elicitation
description A type of pile, placed with a screwing action. Bore piles move earth onto the surface, planning must take into account the removal of this waste.
domain properties
sub-type-of *PILES*

concept *DRIVEN-PILES*
knowledge elicitation
description Hammered into the ground
domain properties
sub-type-of *PILES*

Part-of relationship

The aggregation (part-of) relationship allowed the domain experts' grouping of concepts to be identified. During planning, experts reason at different levels of concept aggregation. For example, activities are created and sequenced at the level of "the foundations" at early points of the planning process, but later refined to the constituent components of the foundations.

Below is the KADS representation of the specific foundations instance occurring in a building. *The-foundations* are shown to have several sub components (*pile1 ... beam5*). A graphical representation is given in Figure 5-9.

instance *the-foundations*
instance of: FOUNDATIONS
implementation: I_Sub Value < *pile1 pile2 pile3 pile4 pile5 pile6 pile7 pile8 beam1 beam2 beam3 beam4 beam5* >

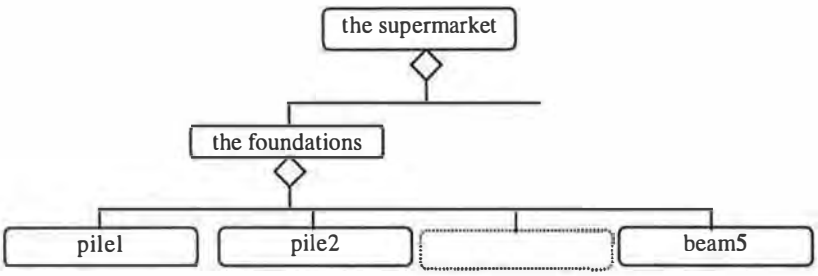


Figure 5-9, Part-of component structure

Domain specific relationships

Whilst the part-of and sub-class relation elicited the structure of the domain's knowledge, domain specific relations were found essential to the planning task. Human planners utilised the interrelationships between components when deciding the ordering constraints between activities. The example *supported-by* relationship is specified below.

```
relation supported-by
        inverse: supports
        argument-1:Object
                role
                cardinality min 1; max 1;
        argument-2: Object
                role
                cardinality min 0; max infinity;
        semantics: argument-1 is physically supported by argument-2
        tuples: examples. beam1 is supported-by pile3
```

A second set of relationships were identified between actions and components. Generically, experts described components as having associated actions. For example, the statement "*a pile requires a pile mat to be laid*" would translate to the following relation.

concept *PILE*	**[has an action]**	**concept** *LAY-PILE-MAT*

The "has an action" relationship was extended to include the "possibly has an action" relationship. This latter case captures the knowledge that some actions are associated with components only if certain criteria are satisfied.

Experts additionally described actions at different levels of abstraction. This levelling resulted in a collective name for the actions which install a component. The term is then used as a high level construct for placing constraints upon all the constituent actions.

The above framework was modelled by relating components to actions through abstract or compound action relationships. Each action which may be associated with a concept was further subdivided into "must" or "infer" relationships, indicating that some actions "must" always be assisted with a component. Whilst others need inference ("infer") to determine if they should be included within a plan.

5.5.6.1.4 Relations between property expressions

Expressions are defined within KADS as statements about the values of the properties of concepts. An example relationship between property expressions provided by Wielinga, Schreiber and Breuker (1993) is that the concept *amplifier*'s *power-button* property being set to *pressed* causes the *amplifier*'s *power* property to be equal to *on*.

Five types of relation between property expressions were identified in the construction domain. Each is described below.

Relationships between a component and actions

Experts associate each component with a set of possible actions for constructing it. For example, the concept *BEAM* is associated with the possible action set *SET-OUT-POSITION, EXCAVATE-BEAM, BLIND-BOTTOM, LAY-FORMWORK, POUR-CONCRETE, CURE-CONCRETE, VIBRATE CONCRETE, LAY-MOULD-OIL,* and *STRIKE-FORMWORK*.

Which members of the potential action set which will be associated with an actual instance of a component is dependant upon the properties of that component and other component instances. For example, the *formwork-type* property of concept *BEAM* causes a *LAY-MOULD-OIL* action and a *STRIKE-FORMWORK* action to be associated if and only if the *formwork-type* property of beam is equal to *custom*.

Dependency relationships between a component's actions

The actual set of actions associated with a component are sequenced according to the values taken by properties of the component to which they are related. For example, an instance of the concept *BEAM* will order its *LAY-MOULD-OIL* action before its *POUR-CONCRETE* action if the beam's property *formwork-type* is set to *custom*.

Dependency relationships between actions from component relations

The actions associated with components are made dependant upon actions associated with other components as a result of the relationships between components. For example, the actions associated with an instance of concept *BEAM* will be made dependant upon the actions associated with a instance of the concept *PILE* if as *supported-by* relationship exists between the *PILE* and *BEAM* instances.

Aggregate conditions

Aggregate conditions are conditions which hold before an action may be executed but the action set which achieves each aggregate conditions is dependant upon the other components within a specific design. For example, the installation of electrical work requires the condition *dry building* to hold. The actions which combine to constitute a *dry building* are variable. If for example a building contains a concrete floor, the *dry building* condition will only hold after the floor has dried. If, however, the floor is of some other construction they completion of the floor will have no effect of the *dry building* condition.

5.6 Summary and conclusions

The construction domain has a limited history within classical planning research. Publications have centred upon Tate's House Building Domain as a demonstration of the capabilities of task refinement planning systems. Therefore, unlike the military evacuation and logistics domains studied by ARPI researchers, further investigation may identify new generic concepts which may be applied to other domains.

The knowledge elicitation described in this chapter applied a combination of structured interviews, observations, and domain document analysis to acquire the underlying domain knowledge. Knowledge modelling was effected through a subset of the KADS methodology's model set. Organisational and application models were constructed to develop an understanding of the construction industry business process and to provide the context in which a commercially placed planning system must operate. A task model was constructed to identify how human planners partition the planning process, facilitating questioning. The domain knowledge level of the model of expertise was constructed to identify the concepts and relationships within the construction domain which underlay the planning process.

The organisational model identified the position of planning within the overall construction process as commencing after a building's design has been finalised and before actual construction commences. The current human centred planning process is limited by the time the process takes, and the high level of skill and experience required by human planners. Any commercially fielded automated planning system must, however, address a number of socio-economical factors. Specifically, the level of current IT expertise in the industry, the concerns about automation leading to robotic working conditions, disbelief at the capability of machines to undertake what is perceived as a complex task, and the need for explanation from an automated system.

The model of expertise identified the components of a building and their interrelationships as the domain knowledge underlying construction planning. Domain experts associate actions with components depending upon the values of a specific instance's properties. Dependency assessment is achieved from three types of domain knowledge. The actions required to construct a component are ordered dependant upon the values of a specific instances properties. The set of actions relating to a component are ordered because of the interrelationships between components. Aggregate conditions are satisfied depending upon the type of components which exist in a specific design.

Blank

6. Limitations of existing representational devices - experiments within the construction industry

In order to draw a limit to thinking, we should have to
be able to think both sides of this limit.
Ludwig Wittgenstein (1889–1951)

6.1 Introduction

This chapter verifies and extends the set of limitations with classical and model-based planning technologies identified in Chapter 4 through a set of encodings of the construction domain elicited in Chapter 5. From the issues identified, a precise rationale for integrating classical and model-based planning is developed.

The classical encoding is divided into two cases: the specific and the generic. The specific case tests the ability of classical planning to represent a specific instance of a building's design. The generic case identifies the issues encountered when providing an encoding of generic planning knowledge which may be applied to a number of specific designs. The model-based encoding considers only the generic case. From the limitations identified, a precise rationale is constructed for integrating classical and model-based planning technologies.

6.2 Classical planning

This section identifies the issues encountered when encoding the construction domain in a task refinement formalism. Two cases are provided. The first considers only a specific building's design, and the second a domain description which may be applied to a number of specific designs.

Before presenting the two encoding cases, the framework followed for translating the KADS model of expertise of the construction domain into a task refinement formalism is described.

6.2.1 Task formalism method

The KADS model of expertise describes the domain knowledge used by human planners in the construction industry. This section considers the methods available to support the translation of this model into a task refinement formalism.

Erol (1995, pp91) summaries the status of the current methods for advising the process of writing planning domains as follows:

"... it is the most neglected aspect of planning, and there is not an established software-engineering methodology to guide this job."

Whilst Erol's comments are valid, the Task Formalism Manual (Tate, Drabble, & Dalton 1994b 1995) provides some considered steps, collectively known as the Task Formalism Method (TFM), to guide the writing of domain descriptions. The authors, however, place the following qualification on the maturity of their method:

"We rather grandly call this the Task Formalism Method (TFM) to reflect our desire to gather experience of writing TF to improve the method itself.
(Tate, Drabble, & Dalton 1994, p 59)

Whilst the task of encoding domains is not yet supported by intensively researched methods, the TFM does encapsulate practical experience. Hence, it is a worthy guiding framework. Each stage of the TFM is briefly outlined below together with details of how it was applied to the KADS model of expertise.

6.2.1.1 TFM step 1: scope of the domain and initial analysis

> *Like any data analysis task, it is important to plan carefully how a domain description is to be provided in TF to O-Plan. It is all too easy to let a domain description grow in a haphazard and inconsistent way... It is useful to view one user role in writing a domain description in TF as being that of Domain Expert. This user will decide on the scope of the domain and introduce the top level description. It is then possible to "fill-in" the details by considering other information given to describe a domain in TF as being provided by one or more Domain Specialist.*
>
> (Tate, Drabble, & Dalton 1994b, pp 59)

Figure 6-1, TFM step 1

Figure 6-1 presents the first stage of the TFM. This step calls for a considered approach to the analysis of a domain. Specifically, it recommends the selection of a single domain expert from whom an outline "skeleton" of a domain may be elicited. Other domain specialists may then be called upon to add detail to this skeleton. Thus, the skeleton provides a structure to a domain within which other expert's knowledge may be positioned.

In the context of the construction domain, the "considered approach" criteria was satisfied through the application of the KADS methodology. KADS provides a considered and structured approach to analysing a domain. The "structure domain and then refine" criteria was also satisfied through the application of KADS. The first stage of constructing an organisation model resulted in a strategic overview of the domain. The organisational model was then refined into application, task, and expertise models by eliciting knowledge from other domain experts at a progressively lower level within the organisation. At each organisational level, experts comments' were positioned into the overall picture derived from the strategic expert. When knowledge elicited at a low level in the domain did not fit into the strategic framework, the high level expert was consulted. The process either identified a flaw in the understanding of overall domain structure or a flaw in the low level knowledge.

6.2.1.2 TFM step 2: action expansion or goal achievement

> *Two different approaches are possible to model domains. A hierarchical action expansion approach is primarily supported by O-Plan. However, it is also possible to state required conditions on the state of the world at certain points - a goal achievement approach...The approaches can be mixed in any way convenient to model the domain. However, it is useful to consider which is to be the main approach during the initial domain modelling exercise.*
>
> (Tate, Drabble, & Dalton 1994b, pp 59)

Figure 6-2, TFM step 2

The second stage of the TFM (Figure 6-2) recommends an early commitment to the primary method of attainment (task refinement or precondition achievement). Both precondition achievement and action expansion are described in detail within Chapter 2.

The decision to base the domain modelling around precondition achievement or task refinement was based on two early observations. First, domain experts spoke in terms of methods for achieving the construction of each component. For example, experts made statements of the form "in order to build the foundations you must in this case carry out the following actions". Second, the potential size of a precondition achievement solution's search space was considered prohibitively large.

Whilst action expansion was selected, the decision was considered reversible if a encoding from this perspective proved unfeasible. However, in the case of the construction industry, the decision proved correct. This observation concurs with, and therefore supports, Drummond's (1994) argument that industrial domains map naturally to task refinement as opposed to precondition achievement representations.

6.2.1.3 TFM step 3: levels of modelling

Step 3 (Figure 6-3) of the TFM provides guidance for writing the actual schemas. The stage provides two key recommendations. First, that the process should proceed by encoding schemas at a high level, before gradually moving down towards the lower levels and ultimately the primitive schemas. Second, that conditions and effects be attributed to modelling levels. The resultant levels may then be used in conjunction with modelling heuristics to ensure consistency between the set of condition types available and the effects within a domain description.

It is all too easy to introduce actions, events, effects and resources and state conditions or use resources at different levels, making the modelling awkward and unnatural. This is sometimes referred to as "hierarchical promiscuity" or "level promiscuity". This will almost certainly lead to the inability to make effective use of search restriction domain information such as condition and resource types.

Actions and the effects they introduce are at a particular domain modelling level. Higher levels are more abstract, lower levels more detailed. In some cases certain (external) types of conditions can only be stated on effects introduced at a domain modelling level which is at a higher or the same modelling level as the condition.

In anything other than trivial domains, it is essential to have a plan based on an initial analysis of the structure of the problem to decide on what actions, events, effects and resources will be modelled at progressively more detailed levels.

1. *Identify the main actions (or events) that will appear at the top of a task or plan. This is the task or top level.*
2. *Gradually worked down through progressively lower levels of detail and try to identify the more detailed actions (and events) to be introduced. It is best if each level introduced has some real meaning to those involved in planning in the real world. Giving a name to each level is a good discipline to ensure that the modelling levels will be useful.*
3. *It is useful to decide on what statement about the world (in the form of effects) will be introduced and manipulated at the various levels by the actions (and events) at each level.*
4. *It is only after these steps have been taken that the conditions required for each actin (or event) need to considered. It is then possible to ensure that these are introduced at a level at or below the level in which the relevant effects are introduced. Type information to restrict the usage of conditions to those that are meaningful in the domain can now be added readily.* (Tate, Drabble, & Dalton 1994b pp 59-60)

Figure 6-3, TFM step 3

The purpose of assigning conditions and effects to distinct modelling levels is to address the problem of hierarchical promiscuity (Wilkins 1988, pp 49 - 57). Problems surrounding hierarchical promiscuity are described by Collins & Pryor (1992; 1994). Tate, Drabble and Dalton (1994a) suggest that adherence to modelling levels will resolve the issues raised by Collins & Pryor.

To address the hierarchical promiscuity issue, the set of criteria defining the relationships between condition types and effects within (Tate, Drabble, & Dalton 1994a) will be followed.

6.2.2 Case 1 - encoding a specific design in a task refinement formalism

6.2.2.1 Introduction

The aim of this case is to test the ability of task refinement planning to represent and reason with the planning knowledge within a specific building's design. This case considers the supermarket extension design as depicted in Figure 6-5 below.

This specific case has limited commercial utility. The construction domain application model, defined in Chapter 5, requires a planning system which can produce plans for a variety of designs. However, this case verifies the ability of task refinement formalisms to represent a building's design: a necessary capability to demonstrate before considering the problem of reasoning with different designs is considered.

This case is similar to Tate's House Building Domain (Tate 1976) in scope and therefore serves to verify that experiment in addition to providing the author with experience of encoding domains in a task refinement formalism.

6.2.2.2 Encoding the actions

Figure 6-5 below presents the components of the specific supermarket extension project detailed in Chapter 5. A subset of the relationships between components are depicted as dotted lines. The encoding problem is to represent in a task refinement formalism the schemas necessary to produce a plan for constructing this building.

The first stage of the TFM (defined in Figure 6-3) is to identify the main planning task. A user of the construction planning system will wish the planner to achieve the task *build supermarket_extension.* This initial task is encoded below in Figure 6-4. The task definition specifies a plan with two dummy nodes (*start, finish*) denoting the initial and final points of the plan. The non-primitive action *build supermarket_extension* specifies the task which the planner must achieve.

```
task build_supermarket_extension;
  nodes  1 start,
         2 action {build supermarket_extension},
         3 finish;
  orderings 1-->2, 2-->3;
end-task;
```

Figure 6-4, Task definition for *build_supermarket_extension*

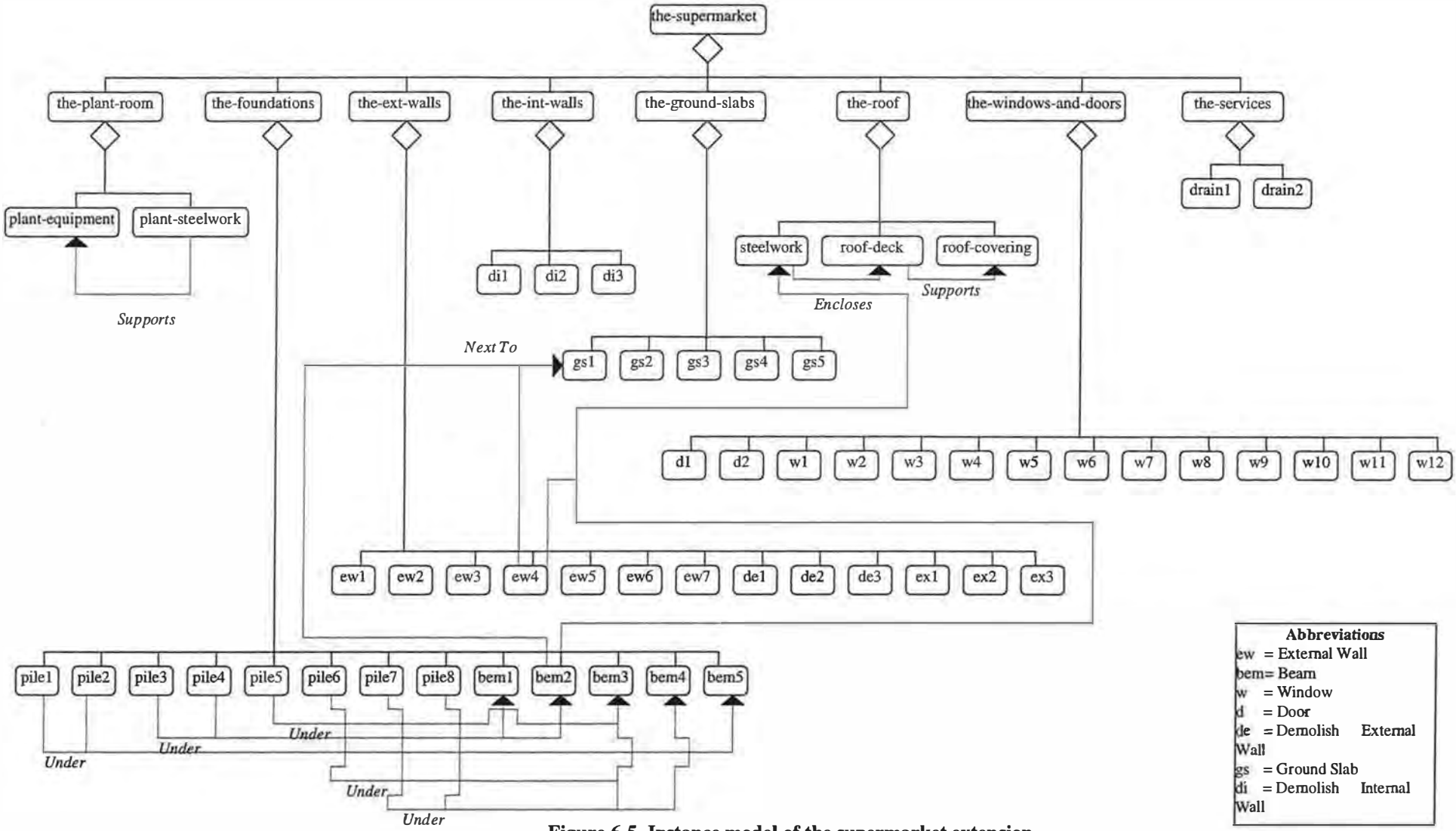


Figure 6-5, Instance model of the supermarket extension

With the problem definition complete, the next encoding stage is the provision of a schema for refining the *build supermarket_extension* task. This schema is depicted in Figure 6-6 below. The schema contains a node for each of the components which are immediate subcomponents of the *supermarket* instance in Figure 6-5.

```
schema build_supermarket_extension;
  expands {build supermarket_extension};
  nodes   1 action {install plant_room},
          2 action {lay foundations},
          3 action {build external_walls},
          4 action {lay ground_slabs},
          5 action {lay roof},
          6 action {install windows_and_doors},
          7 action {install services};
  orderings   4-->1;
end-schema;
```

Figure 6-6, Encoding the schema *build_supermarket_extension*

The single ordering constraint within the schema (*4-->1*) encodes the knowledge that the plant room is composed of heavy electrical equipment (boilers, air conditioning etc.). Hence, the areas construction must not commence until the building's floor is in position, therefore, permitting the delivery of the heavy equipment.

Figure 6-7 below presents the subset of the building for which schemas will be written in the remainder of this section. The *lay roof* node within the schema in Figure 6-6 requires an expansion which includes actions for constructing the subcomponents of *the-roof* component in Figure 6-7. Specifically, the laying of the roof steelwork, the roof deck, and the roof covering. A schema meeting this requirement is depicted in Figure 6-8 below.

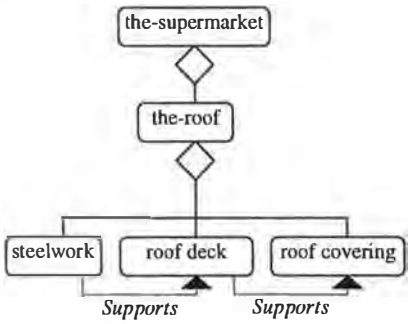


Figure 6-7, Fragment of the building's design

```
schema lay_roof;
  expands {lay roof};
  nodes   1 action {erect steelwork},
          2 action {lay roof_deck},
          3 action {lay roof_covering};
  orderings   1-->2, 2-->3;
end-schema;
```

Figure 6-8, schema *lay_roof*

The relationships between the *steelwork* the *roof-deck* and *the roof-covering* are translated into the ordering constraints *1-->2*, and *2-->3*. This encoding demonstrates the transition from relationships in the domain to ordering constraints in a task refinement formalism.

This encoding process may be repeated for the remaining subcomponents of *the-supermarket*.

6.2.2.3 Encoding the conditions and effects

With the encoding of schemas completed, the TFM advises that next a domain's effects and then conditions be considered. The TFM suggests that a domain is first structured into a number of levels. Figure 6-9 below divides a fragment of the restaurant extension into levels (the thick horizontal lines). The assignment of levels is based upon the different types of component within the domain. The project level component describes the single concept which makes up a project. The project component may have subcomponents but will not itself be a subcomponent of any other component. The aggregate component level consists of the immediate subcomponents of the project level. Components at this level may have subcomponents. Finally, the primitive component level is made up of components which have no subcomponents. This level represents the lowest level of abstraction within the construction domain.

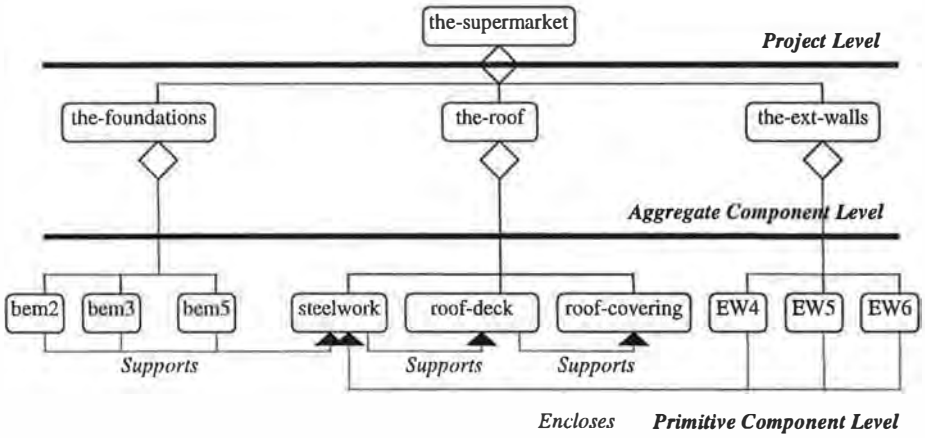


Figure 6-9, Modelling levels attached to the supermarket building

With the domain partitioned into levels, the encoding of effects and then conditions may commence. Actions at the project level result in the single effect *supermarket extension = complete*. The aggregate component level produces the effects resulting from the construction of each aggregate component. From the figure above, the effects *foundations = laid, roof = laid,* and *external walls = built* may be assigned. The primitive component level introduces the effects associated with the completion of the lowest level components within the building. In the figure above, the effects will include *beam2 = laid, beam3 = laid, the roof steelwork = in_position,* etc.

With effects defined and assigned to levels, the encoding of conditions may commence. Tate, Drabble, & Dalton (1994) provide guidance on the relationships between condition types and the level of effects upon which they may be placed. Supervised conditions may only be placed upon effects at the same or lower modelling level. Hence, the lay roof action at the aggregate level may assert the supervised conditions *the_roof_steelwork = in_position* between the primitive level *lay roof_deck* action and *lay roof_steelwork* action. Unsupervised conditions must be placed on effects at the same or higher modelling level. Therefore, the primitive level component *lay roof_steelwork* may place the unsupervised condition *beam1 = laid* which results from the primitive level *lay beam* action The completed schemata are depicted in Figure 6-10 below.

```
schema build_supermarket_extension;
  expands {build supermarket_extension};
  nodes   1 action {install plant_room},
          2 action {lay foundations},
          3 action {build external_walls},
          4 action {lay ground_slabs},
          5 action {lay roof},
          6 action {install windows_and_doors},
          7 action {install services};
  orderings   4-->1;
  conditions
      supervised {ground_slabs laid} at 1   from [4];
  only_use_for_effects
      plant_room  = built at 1,
      foundations = laid at 2,
      external_walls = built at 3,
      ground_slabs = laid at 4,
      roof = laid  at 5,
      windows_and_doors = installed   at 6,
      services = installed at 7;
end-schema;

schema lay_roof;
  expands {lay roof};
  nodes   1 action {erect steelwork},
          2 action {lay roof_deck},
          3 action {lay roof_covering};
  orderings   1-->2, 2-->3;
  conditions
      supervised {steelwork erected} at 2 from [1],
      supervised {roof_deck laid} at 3 from [2],
      unsupervised  {beam2 laid} at 1,
      unsupervised  {beam3 laid} at 1,
      unsupervised  {beam5 laid} at 1;
  only_use_for_effects
      steelwork = erected at 1,
      roof_deck  = laid at 2,
      roof_covering = laid at 3;
end-schema;
```

Figure 6-10, Completed specific case schemata

6.2.2.4 Case 1 - conclusion

The encoding above demonstrates that it is possible to represent a specific building's design within a task refinement formalism. By following the stages of the TFM it is feasible to encode a KADS model of expertise into schemata. Once combined through the task refinement process, the schemata above will define a plan for constructing the restaurant facility.

Whilst the case demonstrates the expressiveness of a task refinement formalism within the context of a specific design, the commercial utility of such a system is limited. The application model developed in Chapter 5 demands a system which can generate plans for a number designs. Case 2, below, considers the issues encountered when producing such an system.

6.2.3 Case 2 - a generic encoding

The aim of this second case is to test the ability of task refinement formalisms to capture generic construction planning knowledge which may be applied to a number of designs. The case meets the requirements of the application model in Chapter 5.

6.2.3.1 Overview of the problem and the encoding approach

The previous case (section 6.2.2) encoded knowledge of the form depicted in Figure 6-11 below. This figure is an instance diagram, with each rounded box representing a specific component within a specific building. In the case of the figure below, the specific building is the supermarket restaurant extension elicited in Chapter 5. Case one demonstrated that task refinement formalisms are capable of capturing this specific case.

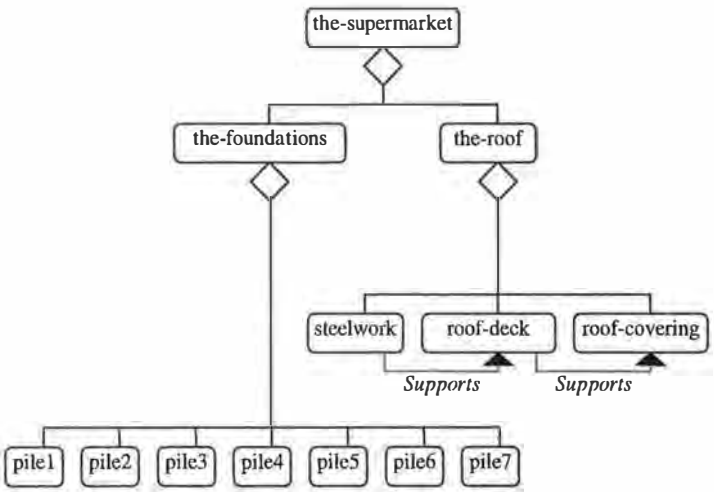


Figure 6-11, fragment of the supermarket extension instance diagram

Figure 6-12 provides a generic representation of construction knowledge which may be instantiated for different buildings. The figure is composed of rectangles, each representing a class of component. The diagram is stating that a building may be made up of one instance of two components: a foundation and a roof. The class *FOUNDATIONS* may be further decomposed into any number of instances of the classes *BEAM* and *PILE*. Instances of the *PILE* class may be supported by any number of instances from the *BEAM* class. Note that the relationships within Figure 6-12 specify the type of relationships which may exist. For example, the diagram is stating that a relationship may exist between instances of the class *ROOF-STEEL-WORK* and the class *ROOF-DECK*, but not between the classes *ROOF-STEELWORK* and *ROOF-COVERING*.

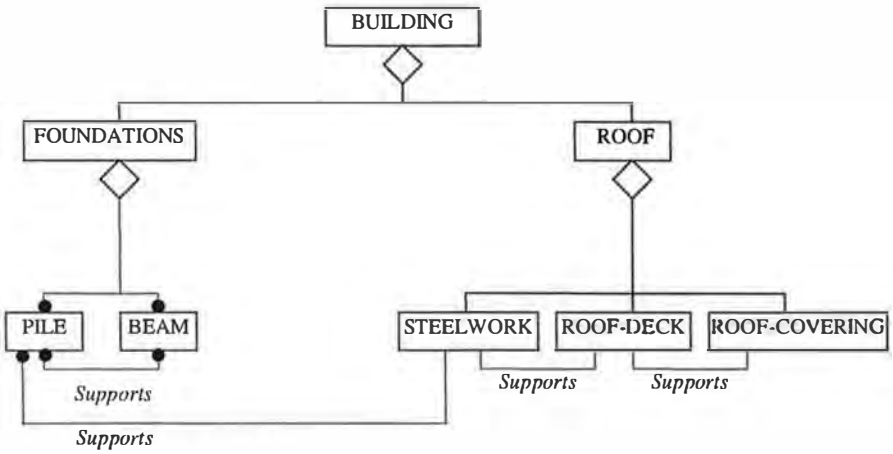


Figure 6-12, Generic model of construction planning knowledge

The generic case requires the capability of specifying generic construction planning knowledge of the form depicted in Figure 6-12. The following sections detail the issues encountered when providing a representation within a task refinement formalism which meets the requirements of the generic case.

The encoding approach follows the TFM guidelines described in the overview at the start of this chapter. First, the main task a planner is to achieve is encoded. From this initial task, a number of refinements are written to generate the actions required by a specific building. With action synthesis considered, the encoding address the issues surrounding conditions, effects, and ordering constraints. Before considering the synthesis of actions, the representation of specific designs is considered.

6.2.3.2 Specifying a specific design in a task refinement formalism

This section identifies how a specific design may be specified within a task refinement formalism.

Classes (e.g. *FOUNDATIONS*, *EXTERNAL-WALLS*) may be translated into types and instances of classes into instances of those types. An example of this translation is depicted below.

```
typesPROJECT          =  supermarket_extension,
        FOUNDATIONS      =  the_foundations,
        PILE             =  pile1, pile2,

        ROOF_COVERING    =  the_roof_covering;
```

In the fragment above, the class *FOUNDATIONS* within Figure 6-12 has been translated into the type *FOUNDATIONS*. The instance of the class FOUNDATIONS (depicted in Figure 6-11), *the_foundations*, has been translated into a instance of the type *FOUNDATIONS*.

The properties of instances may be translated into *initially* statements. A task refinement planner's compiler will note that no actions in the domain's library modify these facts, hence, the statements may be queried without reference to the question and answering system. The computational cost of checking these conditions is therefore inexpensive. The value of a specific instance of class *BEAM*'s formwork type is depicted below.

```
Initially formwork_type beam1 = custom;
```

The structural relation *subcomponent* may be translated into a property of a instance. The example below defines the subcomponents of the instance *building* to be *the_foundations* and *the_roof*.

```
Initially sub supermarket_extension = {the_foundations, the_roof};
```

Domain specific relationships between instances may be translated in the same way as the *sub* relationship. The example below specifies that the instance *beam1* is *supported_by* the instances *pile1* and *pile2*.

```
Initially supported_by beam1    = {pile1, pile2};
```

Using the above encoding it is possible to specify a number of specific designs within a task refinement formalism.

6.2.3.3 Specifying the overall task the planner is to achieve

The user of a construction planning system would wish to specify the task *build ?building*, where the variable parameter *?building* corresponds to the name of the project to be constructed. Figure 6-13 presents the encoding of the task specification for the supermarket extension project. The definition initiates a plan with *start* as the first dummy node, *finish* as the last dummy node, and a single non-primitive task *build supermarket_extension.* The term *supermarket_extension* must match to the term defined as being of type *PROJECT* in the design specified within the domain's *initially statements* (see section 6.2.3.2).

```
task build_supermarket_extension;
  nodes   1 start,
          2 action {build supermarket_extension},
          3 finish;
  ordering
      1-->2,2-->3;
end-task;
```

Figure 6-13, Generic case initial task definition

The design specification and initial task specification should form the only design dependant parts of the generic encoding. With the ability to specify a design and initiate a HTN planning system to plan on that design established, the following sections consider the encoding of schemas for producing the relevant actions, conditions, effects, and ordering constraints.

6.2.3.4 Specifying the *project* level to the *aggregate component level*

Figure 6-9 in the specific case assigned the modelling levels *project, aggregate component*, and *primitive component* to a specific design. The same levels may be assigned to the generic case as depicted in Figure 6-14 below. This section considers the problem of specifying a refinement to the initial task specification which generates the actions required by an instance of *BUILDING*.

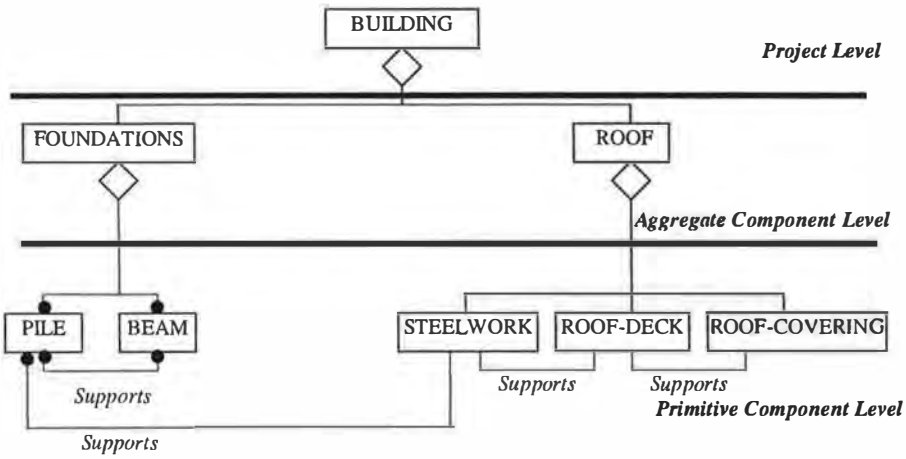


Figure 6-14, Generic model with modelling levels assigned

An instance of the class *BUILDING* will have a set of subcomponents defined. Figure 6-14 specifies what classes these subcomponents may be instances of, specifically, a single instance of class *ROOF* and a single instance of the class *FOUNDATIONS*. An instance of class *BUILDING* may be related to one or both of these classes with instance names defined by the domain writer.

The following *build_building* schema uses the *foreach* construct to generate a *install ?component* action for each subcomponent of a specific building. The *build ?a_building* expands pattern will match against the initial task definition defined above.

```
schema build_building;
    vars ?a_building BUILDING;
    expands {build ?a_building};
    nodes N for each action {install ?component} for
        ?component over {set of subcomponents of ?a_building};
end-schema;
```

Figure 6-15, Schema *build_building*

If a specific building is defined as having the following subcomponents:

```
Initially sub_supermarket_extension    = {the_foundations, the_roof};
```

The *build_building* schema will be instantiated as follows:

```
schema build_building ;;;instantiated
    vars ?a_building BUILDING;
    expands {build supermarket_extension};
    nodes  1 action     {install the_foundations},
           2 action     {install the_roof};
end-schema;
```

The *build_building* schema successfully accounts for number and names of components within a specific design.

6.2.3.5 Specifying the refinements within the *aggregate component level*

The example within Figure 6-14 contains only one level of components at the aggregate component level. The classes *FOUNDATIONS* and *ROOF* are composed of primitive components. Within a building, it is common to have components which refine through several levels of aggregate components before the primitive level is reached. Consider the example in Figure 6-16 below. The figure is stating that a building may have a subcomponent *WALLS* which is in turn decomposed into *INTERNAL-WALLS* and *EXTERNAL-WALLS*.

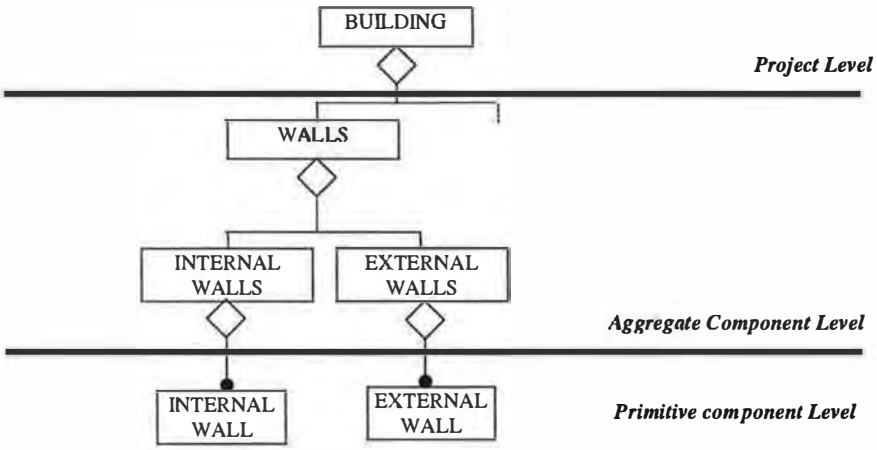


Figure 6-16, Example of multi levelled aggregate components

The *project* to *aggregate component level* encoding mechanism described in section 6.2.3.4 will produce a task *install the_walls* if and only if an instance of the class *BUILDING* is defined as having a subcomponent which is an instance of class *WALLS*. The task of the aggregate to aggregate level encoding is to expand the *install the_walls* task to account for instances of classes *INTERNAL-WALLS* and *EXTERNAL-WALLS*.

The *aggregate* to *aggregate component level* encoding may be achieved using the same *foreach* construct method as in the *project* to *aggregate component level* encoding schema. The *install_walls* schema below will generate an *install ?a_set_of_walls* task for each instance of classes *INTERNAL-WALLS* and *EXTERNAL-WALLS* defined as subcomponents of an instance of class *WALLS*.

```
schema install_walls;
    vars ?a_set_of_walls : WALLS;
    expands {install ?a_set_of_walls};
    nodes  N for each action {install ?component} for
        ?component over {set of subcomponents of  ?a_set_of_walls};
end-schema;
```

Assuming a specific design includes the instances *the_internal_walls* and *the_external_walls* defined as subcomponents of an instance of class *BUILDING*, the following schema will be instantiated.

```
schema install_walls ;;;;;instantiated
    vars the_walls : WALLS;
    expands {install the_walls};
    nodes   1 action {install the_internal_walls},
            2 action {install the_external_walls};
end-schema;
```

The encoding above successfully generates an appropriate number of actions when moving between layers of the *aggregate component* level.

6.2.3.6 Specifying the refinement from the aggregate component level to the primitive component level

The sections above describe how an a task refinement planner may successfully capture the refinements from the *project level* to the *aggregate component level* and from *aggregate component level* to *aggregate component level.* This section considers the penultimate refinement from *aggregate component* level to *primitive component level.* As demonstrated in Figure 6-17, this transition may be achieved using the same *foreach* mechanism as in the previous model levels.

```
schema lay_foundations;
    var ?a_set_of_foundations : FOUNDATIONS;
    expands {install ?a_set_of_foundations };
    nodes   N for each action {install ?component} for
            ?component over {set of subcomponents of ?a_set_of_foundations};
end-schema;
```

Figure 6-17, Example transition from aggregate to primitive level components

If the *lay_foundations* schema is applied to a instance of class *FOUNDATIONS* which is related to the subcomponents *beam1, beam2, pile1,* and *pile2*, the schema will be instantiated as follows:

```
schema lay_foundations;; instantiated
    var the_foundations : FOUNDATIONS;
    expands {install the_foundations};
    nodes   1 action {install beaml },
            2 action {install beam2},
            4 action {install pile1},
            5 action {install pile2};
end-schema;
```

Task refinement formalisms may successfully capture the transition from aggregate to primitive level components.

6.2.3.7 Specifying the refinements at the *primitive component level*

The encoding described in the sections above traverses a building's design from the task level to the primitive component level accounting for the variable number of components which may exist in a specific design. The *primitive component level* encoding must account for the actual actions required to construct each primitive component. For example, the schema in Figure 6-17 will produce a number of *lay pile* and *lay beam* tasks which must each be refined to include within a plan the actions required to construct them. This section considers the issues surrounding such an encoding.

Taking the refinement of a *lay beam* task, there are a number of possible refinements, two methods from this set are depicted below. The applicability of the methods is distinguished by the type of formwork used within a beam. If a beam has prefabricated formwork the first method must be employed. If the beam has custom formwork then the second method is employed. This distinction is made through the bolded *only_use_if* filter conditions.

```
schema lay_beam_prefabricated_formwork;
  vars ?beam : beam;
  expands (lay ?beam);
  nodes   1 action (set_out_position ?beam),
          2 action (lay_formwork ?beam),
          3 action (lay_steelwork ?beam),
          4 action (pour_concrete ?beam);
  only_use_if ?beam formwork = prefabricated;
end-schema;

schema lay_beam_custom_formwork;
  vars ?beam : beam;
  expands (lay ?beam);
  nodes   1 action (set_out_position ?beam),
          2 action (lay_formwork ?beam),
          3 action (lay_steelwork ?beam),
          4 action (pour_concrete ?beam),
          5 action (strike_formwork ?beam);
  only_use_if ?beam formwork = custom;
end-schema;
```

The *primitive component level* encounters the same encoding issues described within the encoding of the task *obtain permission to build* in Chapter 4. Chapter 4 considered the case of a single condition, multiple conditions effecting the selection of methods for achieving a task, and the expressiveness of filter conditions. The construction industry problem verifies the conclusions reached within that chapter. The extensions to Tate's House Building domain are realistic.

6.2.3.8 Encoding effects

The encoding method above will generate the appropriate actions for a design. This section considers the issues encountered when adding action effects to this encoding.

Consider the building design depicted in Figure 6-14 and the first level refinement schema *build_building* in Figure 6-15. The *build_building* schema generates the actions required to construct a building and may therefore be assigned the effect *?a_building construction = completed*, where *?a_building* is a variable term which is instantiated to the name of a specific building. The *build_building* schema with this effect attached is depicted below.

```
schema build_building;
    vars ?a_building BUILDING;
    expands {build ?a_building};
    nodes N for each action {install ?component} for
        ?component over {set of subcomponents of ?a_building};
    only_use_for_effects
        ?a_building construction = completed;
end-schema;
```

It is not possible to include effects within the *build_building* schema which are dependant upon the actions it generates. For example, over a specific design the schema may generate the actions *install the-foundations,* and *install the-roof.* Task refinement formalisms do not support constructs to permit the effect *foundations = laid,* and *roof = built* to be synthesised within the *build_building* schema. These effects must be asserted by the relative expansions of *build_building* (see schema *lay_foundations* below).

```
schema lay_foundations
    vars ?a_foundation : FOUNDATIONS;
    expands {install ?a_foundation};
    nodes N for each action {install ?component} for
        ?component over {set of subcomponents of ?a_foundation};
    only_use_for_effects
        ?a_foundation = laid;
end-schema;
```

The same encoding mechanisms may be used to successfully encode action effects from *project level* through to *primitive level* components.

6.2.3.9 Generating conditions from relationships

Relationships between components are the main cause of ordering constraints between actions in the construction domain. Figure 6-18 presents a fragment of an object model depicting the possible relationships between components. The figure is, for example, stating that an instance of the class *STEELWORK* may be related to a number of instances of the class *PILE* through the *supports* relationship.

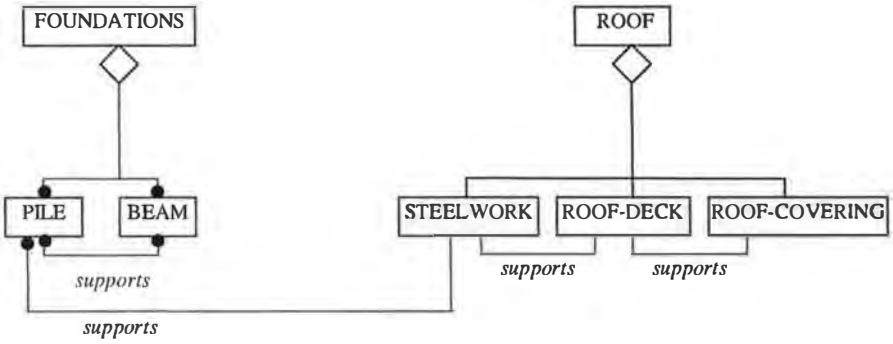


Figure 6-18, Example of relationships between components

Using the design fragment in Figure 6-18, the following subsections consider three cases of relationships causing ordering constraints or conditions between actions. The first case considers a number of related components which share the same immediate super component. The second case considers a number of related components which do not share the same immediate super component. Finally, the third case considers relationships which do not automatically imply ordering constraints between actions.

6.2.3.9.1 Related components which share the same immediate super component

Consider the *PILE* and *BEAM* classes in Figure 6-18 above. The figure is stating that instances of class *PILE* and class *BEAM* may be related through a *supports* relationship. The semantics of this relationship are that the *PILE* instance is physically supporting the weight of the *BEAM instance*. Hence, the *PILE* instance must be constructed before the *BEAM* instance.

Actions for specific *PILE* and *BEAM* instances are first introduced at the *aggregate component level* to *primitive component level* stages of an encoding. Figure 6-17 above encodes these actions and is therefore reproduced below.

```
schema lay_foundations;
    var ?a_set_of_foundations : FOUNDATIONS;
    expands {install ?a_set_of_foundations };
    nodes   N for each action {install ?component} for
        ?component over {set of subcomponents of ?a_set_of_foundations};
end-schema;
```

Figure 6-19, Reproduction of Figure 6-17 - schema *lay_foundations*

The schema will work over each subcomponent of an instance of the class *FOUNDATION* generating an install action. Assume that a design contains an instance of class *FOUNDATIONS* named *the-foundations*, an instance of class *PILE* named *pile1*, and an instance of class *BEAM* named *beam1*. The subcomponents of *the-foundations* are *pile1* and *beam1* and *pile1* is *supporting beam1*. Over this design, schema *lay_foundations* will be instantiated as follows:

```
schema lay_foundations ;; instantiated
    vars the-foundations :FOUNDATIONS;
    expands {install the-foundations};
    nodes   1 actions {install pile1},
            2 actions {install beam1};
end-schema;
```

With schema *lay_foundations* instantiated, it is desirable to add a *supervised* conditions supported with an ordering constraint to capture the fact that the *install pile1* action should be completed before the *install beam1* action because of the *supports* relationship between the two instances. A *supervised* condition of the form *pile1 laid at 2 from 1* and the ordering constraint *1-->2* would permit the HTN planner to maintain this constraint without need to discover how to establish it.

However, task refinement formalisms do not provide constructs for adding conditions and ordering constraints dependent upon the actions generated by a *foreach* or *iterate* construct. Consider the *lay-foundations-ideal* schema below, where the bolded lines indicate the type of construct required.

```
schema lay_foundations_ideal;
    var ?a_set_of_foundations : FOUNDATIONS;
    expands {install ?a_set_of_foundations};
    nodes   N for each action {install ?component} for
            ?component over {set of subcomponents of ?a_set_of_foundations};
            ;;;if ?component is of the type BEAM and it is related to any
            ;;;instances of the type PILE through a supports relationship, then
            ;;;include an ordering constraint "all instances of class PILE related
        ;;; --> ?component"
    only_use_for_effects
        foundations laid = true;
end-schema;
```

As it is not possible to include a *supervised* condition within the *lay_foundations* schema, the next option is to consider an *unsupervised* condition within the refinement of a *lay beam* schema. Consider the following *lay_beam_prefabricated_formwork* schema.

```
schema lay_beam_prefabricated_formwork

    vars ?beam : beam;
    expands (lay ?beam);
    nodes   1.action (set_out_position ?beam),
        2.action (lay_formwork ?beam),
        3.action (lay_steelwork ?beam),
        4.action (pour_concrete ?beam);
    only_use_if ?beam formwork = prefabricated;
        ;;;if ?beam is related to any instances of the type PILE through a
        ;;;supports relationship, then include an ordering constraint "all
    ;;;;instances of class PILE related --> ?component"
end-schema;
```

The bolded lines indicate the type of construct required but not supported by task refinement formalisms for adding an *unsupervised* ordering constraint to a *lay beam* schema if and only if the beam within the schema is supported by a pile.

The above schemas demonstrate that it is not possible to include ordering and condition constraints within task refinement schemas which are dependant upon the ordering constraints between a design's components

6.2.3.9.2 Related components which do not share the same immediate super component

Consider the *PILE* and *STEELWORK* classes in Figure 6-17 above. Instances of class *STEELWORK* may be supported by instances of class *PILE*. Semantically the weight of the *PILE* instance is physically supporting the weight of the *STEELWORK* instance. This case differs from the *PILE* and *BEAM* relationship as the *PILE* and *STEELWORK* classes do not share the same immediate super component. Class *PILE* is a subcomponent of class *FOUNDATIONS* and class *STEELWORK* a subcomponent of class *ROOF*. As the classes do not share the same immediate super component, the encoding described in the sections above will introduce the *lay pile* and *lay steelwork* action in different schemas.

The encapsulation constraint on *supervised* conditions prohibits the constructs use on actions which do not appear within the same schema. Hence, it not possible to place a *supervised* condition between *lay pile* and *lay steelwork* actions. The remaining condition types available to a domain writer are *unsupervised* and *achieve*.

```
schema build_external_walls
  vars ?entity external_walls
  expands install ?entity

  N for each action {install ?type) for
      ?install over (set of items declared as sub of foundations)
          ;;;if install is related to a beam through a supported by
      ;;;relationship then include the following condition
          ;;;unsupervised beam related through = laid at current
          ;;; action
  only_use_for_effects
      external walls built = true;
end-schema;
```

There is currently no construct within the Task Formalism to support this type of reasoning. Therefore, conditions depending upon relationships cannot be implemented.

6.2.3.9.3 Case when a relationship between components does not automatically imply dependency constraints

Consider the case of a drain being related to a beam by an *on top of* relationship. The semantics are that the drain runs below the intended location of a beam. If the beam is laid first, a tunnel must then be dug under the beam to allow the drain to be placed. If the drain is laid first, the beam may be laid over the drain. Hence, it is simpler to lay a drain before any beam which is related to it through an *on top of* relationship.

A drain may not always be laid before the beam it passes under. For example, the beam may require mechanical devices to support its positioning. If the drain is laid first there is a danger of the mechanical devices damaging the drain. Thus, inference is required to determine if an *on top of* relationship between a drain and a beam results in a dependency relationship. There is currently no mechanism in the task formalism to support this reasoning.

6.2.3.10 Generating aggregate conditions

Aggregate conditions are conditions which may be referred to as a single concept, e.g. *building = dry*, but require a variable number of other conditions to hold before they are satisfied. Consider the example of the condition dry building. If a building includes a concrete floor, the building will not be dry until the floor is dry. Thus, the condition *building = dry* will require *floor = dry*. However, if the building does not contain a concrete floor the *building = dry* condition will not require any reference to a *floor = dry* condition.

A current implementation of an aggregate condition is specified below. The two schemas are taken from the O-Plan test set's three little pigs domain.

```
task build_cheap_secure_house;
 nodes   1 start,
       2 finish,
       3 action {build house},
            4 action {check security};
 orderings 1 ---> 3, 3 ---> 4, 4 ---> 2;
 resources consumes {resource money} = 0 .. 500 pounds overall;
end-task;

schema security_checker;
 expands {check security};
 local_vars ?material = ?{type material};
 conditions unsupervised {proof_against wolf ?material},
            unsupervised {material wall} = ?material,
            unsupervised {wolf_proof door},
            unsupervised {wolf_proof windows};
end-schema;
```

The *build_cheap_secure_house* task generates two non-primitive tasks: *build house* and *check security*. The security check is performed after the house building actions and must be satisfied before planning may be completed. The security checker schema's semantics are as follows. For a building to be wolf proof, the material used in the houses walls construction must be wolf proof and the doors and windows must be wolf proof.

The encoding of a security checker may be applied to the dry building condition.

```
schema dry_building_checker;
  expands {check dry_building};
  conditions
  unsupervised {concrete floor} = dry,
          unsupervised {windows and doors} = fitted;
  only_use_if building contains a subcomponent concrete floor.
end-schema;

schema dry_building_checker;
  expands {check dry_building};
  conditions
          unsupervised {windows and doors} = fitted;
  only_use_if building does not contain a subcomponent concrete floor.
end-schema;
```

The different cases which constitute a dry building are encoded as separate schemas. Which definition of dry building to apply is determined by the *only_use_if* filter condition. Any schema requiring a dry building may therefore include a *check dry_building* task and order it before the actions which require the condition.

This checker schema solution to the aggregate condition issue results in a schema required for each case of what may constitute a dry building. Thus, redundancy is too easily encoded causing a high maintenance overhead.

6.3 Model-based planning

6.3.1 Scope

This section considers the encoding of the generic construction planning knowledge and the translation from a specific design into a plan. The scope is identical to the generic case considered in the task refinement encoding described in section 6.2.3.

6.3.2 Encoding specific and generic knowledge

Model-based planning is centred around a frame based modelling scheme which may be mapped directly onto the KADS model of expertise. Figure 6-20 presents a fragment of generic construction planning knowledge encoded in a model-based formalism. Whist model-based planning uses a different notation to the KADS model of expertise, the mapping of concepts and relationships is direct.

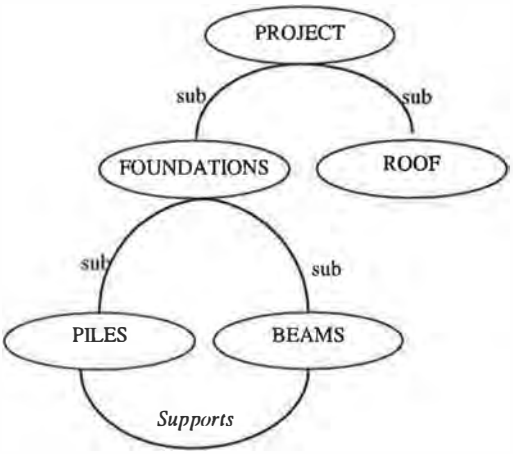


Figure 6-20, Model-based representation of the construction domain

A specific design may be encoded by attaching components to classes. For example, a design which contained two piles will be represented by attaching two instances (*pile1* and *pile2*) the *PILE* class in Figure 6-20.

The following sub sections detail how the model-based representation above may be used to generate actions and ordering constraints.

6.3.3 Generating a variable number of actions

Consider the fragment of the supermarket extension encoded in the model-based formalism in Figure 6-21 below.

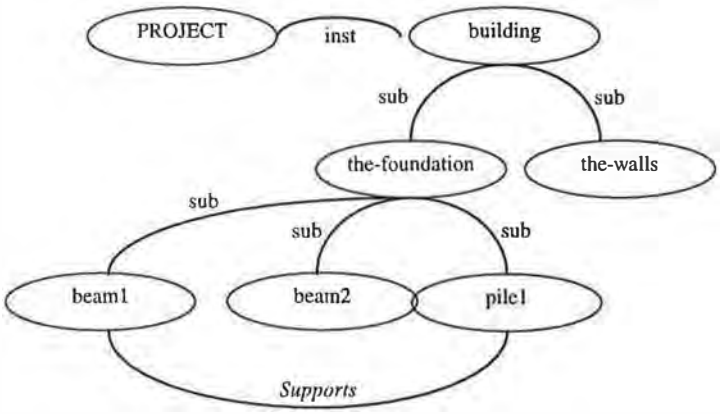


Figure 6-21, Specific building design encoded in the Model-Based representation

The model-based planning algorithm exploits the subcomponent relationships ensuring each object is visited and actions generated. Hence, the planner automatically adjusts the number of actions included within a plan to the number of components in a specific building.

6.3.4 Generating different methods

Model-based formalisms permit actions to be associated with components under two directives: *must*, and *infer*. Must implies that the action will always be associated with a component. *Infer* implies that some inference is required. Consider the encoding of a beam's actions below.

```
BEAM.actions =   must(a, set_out_position),
                 must (a, lay_formwork),
                 must (a, lay_steelwork),
                 must (a, pour_concrete),
                 infer (a, strike_formwork, INFER_STRIKE_FORMWORK)
                 infer (a, pump_area, INFER_PUMP_AREA)
```

The first 4 actions (*set_out_position* to *pour_concrete*) must all be included in a plan which includes a beam. The *strike_formwork* and *pump_area* require inference to determine if they are required. The capitalised statement immediately after the names of these actions in the encoding above indicates the inference mechanism which should be invoked to determine the action's relevance. An example of the inference identified as *INFER_STRIKE_FORMWORK* is encoded below.

```
INFER_STRIKE_FORMWORK
    goal strike_formwork
    Backward chain
    rule 1
    if beam. formwork type = custom then
        true
    end rule
    rule 2
    if beam. formwork type = prefabricated then
        false
    end rule
```

The two rules in the rule set encode the knowledge that a strike formwork action is only required if a beam is made with custom formwork. Thus, it is possible to generate actions for constructing a components depending upon the properties of a component.

6.3.5 Generating dependency from relationships

The model-based formalism permits actions to be ordered according to the relationships between instances. Figure 6-22 below depicts a beam related to a pile through a *supported by* relationship. The semantics of the relationship are that the pile is supported by the beam, hence, the beam should be constructed after the pile. The box next to the supported by relationship represents the reasoning which may be invoked to determine dependency between the components participating in the relationship. The contents of the greyed box are specified below the figure.

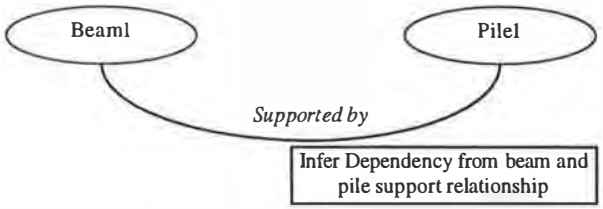


Figure 6-22, Model-based relationship example

```
BEAM.generate dependency
    forall X, where X is supported by self
        make the actions of X dependent upon the actions of self.
    end forall
```

The inference attached to the relationship causes the construction action of the beam to be deponent upon the pile. Thus, it is possible to generate dependency between actions which is dependent upon the relationships between components in a design.

6.3.6 Generating conditions and effects

Model-based planners have not, to date, implemented the concept of action conditions and effects. Dependency constraints may only be reasoned over components which are related through domain specific relationships (e.g. *supported by*).

The absence of action conditions and effects leads to a number of limitations. First, dependency knowledge must be specified in terms of the producing and consuming actions. It is not possible to state an action requires a condition, leaving the selection of producing action to the planning engine. Second, model-based planners cannot identify action interactions and take steps to remove such plan flaws. Action iterations occur when an action deletes the effect of a second action before a third action which requires the deleted effect. Such a functionality requires action conditions and effects combined with a question and answering system (as developed by (Tate 1977)).

6.4 Summary and conclusions

This chapter identified a number of limiting issues encountered when encoding the construction domain in both classical and model-based planning technologies.

Considering the approach to the classical encoding, whilst the process has not been extensively researched, the Task Formalism Method (TFM) provided an effective interface between the KADS model of expertise developed in Chapter 5 and task refinement schemas. The approach of first writing actions at the highest level of abstraction before progressively moving to lower levels of abstraction mapped onto the top down structure of the KADS model set. Writing action effects and then conditions after the action hierarchy has been constructed, combined with the use of modelling levels and the associated levelling constraints on condition types, allowed a considered representation to be developed which avoided the problem of hierarchical promiscuity.

The first classical case considered the problem of specifying the representation required by a specific design. This scope is identical to Tate's House Building domain. The case confirmed that a task refinement formalism may successfully capture a specific design, and provided the author with experience of task refinement encoding. The commercial utility of such an encoding is, however, limited. The application model presented in Chapter 5 demands an encoding which will accept a number of building designs and then synthesis plans based upon the specific components and relationships within each design.

The second classical case considered the requirements of a commercial construction planning application. The case identified that design details may be specified within a planner's always context (facts which do not change during planning), translating classes from the KADS model into types and instances of classes into instances of types. The subcomponent relationship, domain specific relationships, and properties may be converted into logical statements.

The *foreach* construct may be exploited to navigate through the design specified in a planner's always context generating actions according to the number and type of components. This encoding identified two limitations.

First, the issues identified in Chapter 4 surrounding the need to specify multiple methods for achieving tasks distinguished by filter conditions was encountered and confirmed. Methods within the construction domain were found to be dependant upon multiple conditions. The limitations of the task refinement formalisms forced multiple methods to be specified with the associated redundancy and therefore maintenance problems.

Second, the *foreach* construct is not supported with a conditional effect and condition mechanisms. Hence, it is not possible to generate actions and then infer the ordering and supervised or unsupervised conditions which must exist between them. The problem is compounded by the encapsulation constraint between schemas. Even assuming the existence of a conditional condition construct, it would be possible only to generate supervised conditions between components which share the same immediate supercomponent, as actions relating to both components will be generated within the same schema. If two components do not share the same immediate supercomponent they will be generated within different schemas, hence, as the supervised condition construct may only be placed within a schema, the construct may not be used in this case.

Considering the model-based case, only the commercial application requirement was assessed. The constructs within the KADS model of expertise of the construction problem mapped directly onto to model-based representation. Model-based planning provided facilities for generating actions appropriate to the number and type of components in a specific design. The absence of action conditions, effects, and ordering constraints prevent model-based planners from inferring ordering constraints other than those where both the producing and consuming action are specified. Without conditions, effects, and a question and answering algorithm model-based planners cannot detect and resolve interactions between actions.

In conclusion, the generic construction industry case identified limitations with both classical and model-based technologies. The classical limitations are summarised under the following three headings:

- **Expressiveness**. Task refinement formalisms do not provide constructs for inferring conditions and ordering constraints from domain expert knowledge. As a result, first multiple methods must be specified for refining tasks. Second, when applying the *foreach* or *iterate* constructs to generate an appropriate number of actions to match the specific components within a design, conditions resulting from the relationships between components cannot be inferred.

- **Redundancy.** The absence of conditional ordering constraints and conditions force domain descriptions which contain multiple methods for achieving tasks. It is common to find redundancy between these methods in terms of the same actions, ordering constraints, conditions, and effects being repeated a number of times.

- **Semantic distance.** The absence of constructs for deriving conditional ordering constraints and conditions combined with the distance between the criteria within a domain for selecting methods and the filter condition constructs lead to a significant semantic gap between domain and task refinement representation.

The limitations with model-based planning identified are summarised below. Both limitations arise from the absence of the concepts of action conditions and effects within the technology.

- **Automatic establishment of conditions.** Model-based planners may only infer dependency information between components which are explicitly related through domain specific relationships. It is not possible to specify an action conditions, leaving the selection of establishing action to the planning system.

- **Detection and resolution of action interactions.** Without action conditions, effects, and a question and answering algorithm it is not possible to detect and therefore resolve interactions between actions of the form of action effects being deleted before the consuming action may use them.

The limitations with classical and model-based technologies either identified or confirmed through the construction industry encoding provide the following precise rationale for integrating the two technologies.

Model-based planning is designed to exploit domain expert knowledge outside the space of a partial-order state based world model. The technology's representation maps closely to the results from current object based methodologies. In contrast, the representation supported by classical task refinement planners is difficult to map to the reasoning demanded by industrial domains for both determining the actions required within a plan and the ordering constraints between those actions. Classical planners do, however, provide a powerful question and answering facility for both establishing and maintaining conditions over a plan.

An integrated architecture may exploit the representation and reasoning mechanism provided by model-based planning for determining actions and ordering constraints within a plan. A task refinement component may then be used to combine these constraints into a consistent plan.

Blank

7. Development of an integrated architecture

Together we stand. Divided we fall

Proverb

7.1 Introduction

Chapters 4 and 6 identified complementary strengths between the capabilities of model-based and classical planning technologies. This chapter describes an integrated architecture which exploits each technology's relative capabilities. An evaluation of the resultant architecture is detailed in Chapter 8.

The integrated architecture is composed of five components: a set of domain modelling constructs, a model-based planner, a model-based - task refinement planner interface, and a task refinement planner. This chapter describes each component before their functionality is described thorough an example problem from the construction domain (elicited in Chapter 5).

7.2 Overview of the proposed integrated architecture

The proposed integrated architecture is depicted in Figure 7-1 below.

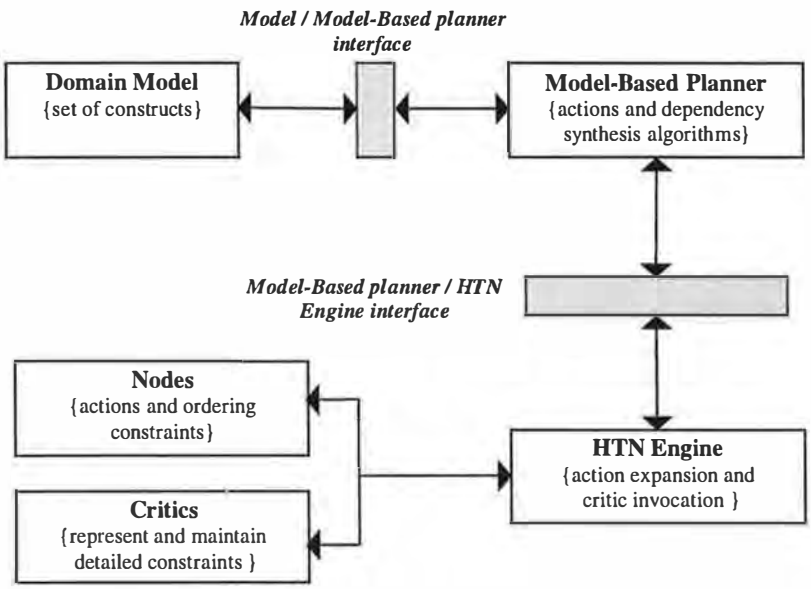


Figure 7-1, Proposed integrated architecture

The domain model provides a set of constructs for representing domain specific knowledge. The model-based planner applies the knowledge specified in the model to generate a set of actions and dependency constraints. The HTN Engine combines the action and dependency knowledge from the model-based planner to create a complete plan. HTN critics are invoked to ensure the establishment and maintenance of conditions and effects over the plan.

The proposed architecture exploits the expressive power of model-based planning for determining the actions and ordering constraints required by a plan. The HTN engine's refinement process and critics assemble the action and dependency information from the model-based planner into a completed plan. This architecture addresses the relative strengths identified in the previous chapters of this thesis.

An important design decision was the isolation of the model-based and HTN planning components via the *MBP / HTN engine interface*. This design permits the architecture to be used as an integrated MBP / HTN planner, a HTN planner with access to both MBP generated schemas and traditionally encoded schemas, or a HTN planner with access only to traditionally encoded schemas. The three modes facilitate experimentation with different methods of generating or encoding schemata for a HTN engine and identification of the different domain features that affect the utility of each mechanism. This facility is exploited within Chapter 8 when addressing the Pacifica domain.

7.3 Domain modelling constructs

This section describes the set of constructs for modelling a domain. The implementation of an inference engine to apply the constructs is described in section 7.4. This section concludes with the encoding of an example problem from the construction industry domain elicited in Chapter 5.

7.3.1 Concepts and actions

Model-based planning (MBP) is based on the axiom that an activity may be modelled as a union between a concept, an action, and a number of resources (Marshall 1988, pp 37). For example, the action *print feasibility study* may be described as the union of the object *feasibility study*, the action *print*, and the resources *printer* and *paper*. The MBP definition of an activity within this chapter is simplified by ignoring the issue of resources. The issues surrounding the inclusion of resources within the modelling scheme is described within the further work section of Chapter 9.

Within the object-oriented paradigm, the union of objects and actions to form the components of an activity may be modelled as a relationship between a class *ACTION* and a class *OBJECT*. Classes provide the template from which actual instances may be created. Figure 7-2, below, depicts the relation *action* between class *ACTION* and class *OBJECT* and an instance of this structure.

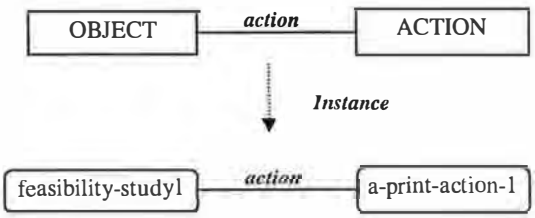


Figure 7-2, Object-oriented representation of objects and actions

Object-oriented modelling supports a specialised relation between classes termed *inheritance*. Inheritance permits a domain to be specified in terms of specialisations. Thus, commonalties between objects and actions may be identified and represented together, whilst differences may be explicitly modelled as specialisations of generic concepts. An example action and object hierarchy is depicted below in Figure 7-3. Inheritance relationships are identified with a triangle notation.

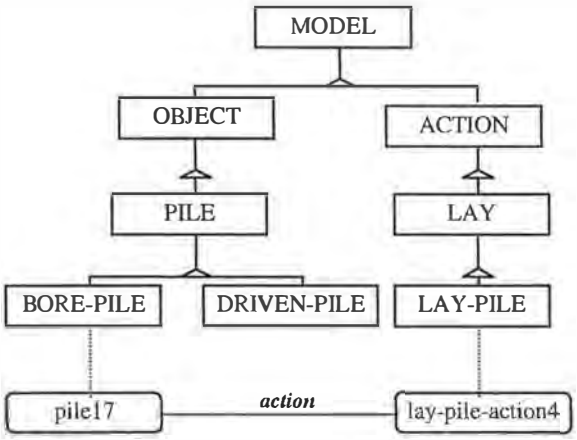


Figure 7-3, Example action and object hierarchy exploiting inheritance

Inheritance is a directional relationship. The point of the triangle notation identifies the *superclass* of an inheritance relationship, whilst the triangle's bottom identifies the *subclass*. Knowledge specified on the *superclass* is copied to the *subclass*. In Figure 7-3 above, class *MODEL* is the superclass of all of the other classes in the figure i.e. a subclass inherits not only from its immediate superclass, but all the superclasses of its superclasses etc.

In Figure 7-2, knowledge common to both types of pile is specified within class *PILE*. Knowledge specific to either bore or driven piles will be specified on the respective class. For example, both driven and bore piles require their position to be set out. This knowledge will be specified on class *PILE* and inherited by both subclasses. Only bore piles deposit quantities of earth on the surface and require infrastructure to remove this earth. The infrastructure knowledge for removing earth is specified only on class *BORE-PILE*.

The organisation of knowledge facilitated by inheritance simplifies the task of maintaining a knowledge base. By modelling knowledge at the most general level possible, modifications can be made in one place, but effect a number of other classes. In the example above, editing general pile knowledge within class *PILE* will update both the *BORE-PILE* class and the *DRIVEN-PILE* class.

7.3.2 Action levels

MBP offers two categories of actions with the aim of synthesising plans for different types of audiences: *abstract* and *primitive*. Figure 7-4, below, presents an object instance diagram with a number of abstract and primitive actions associated.

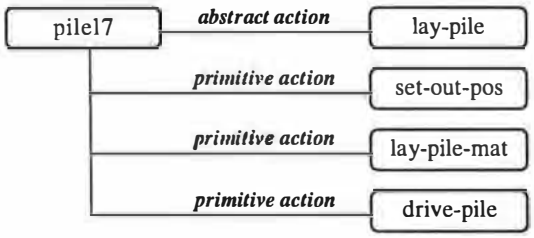


Figure 7-4, Abstract and primitive actions

Within existing MBP implementations, planning is performed in one of two modes: *abstract* or *primitive* (Marshall 1988). In abstract mode, the planner will only instantiate an object's abstract actions, conversely in primitive mode, the planner will only instantiate an object's primitive actions. From the example in Figure 7-4, in abstract mode, the planner will produce a plan which included a *lay pile* action. In primitive mode, the system would include the required members of the set: *set out position, lay pile mat* and *drive pile.*

This implementation of action levelling is flawed. Plans synthesised at the abstract level will not consider the potential constraints between primitive actions. For example, two abstract actions $abstract_1$ and $abstract_2$ may be left unordered relative to each other. If planning had proceeded to the primitive level, however, an ordering constraint may have been identified. Thus at one level of abstraction the planner would not find ordering constraints, whilst at another it may. A more considered planning process should plan to the lowest level of detail available, and an aggregation of that plan produced for different audience levels. Winstanley and Hoshi (1993) demonstrate how to present plans with abstract actions maintaining the detailed constraints of the lower level actions they represent.

Within the MBP component of the integrated architecture, primitive actions are viewed as refinements of an object's abstract actions. This modelling correlates to the NOAH classical planning system (Sacerdoti 1977). Action levelling is included to allow constraints to be placed and actions at different levels of abstraction. Constraints placed upon an abstract action must be maintained by that abstract actions refinements.

7.3.3 Domain model structure

The MBP paradigm provides the following generic model for structuring domain descriptions:

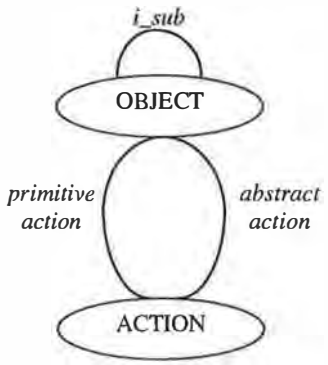


Figure 7-5, Original MBP generic project model (Marshall 1988, pp 48)

Since the model was originally drawn, object-oriented notations have been refined. Figure 7-6, below, presents the original MBP model with the constraints on the structure of the model made explicit through current object-oriented modelling notation.

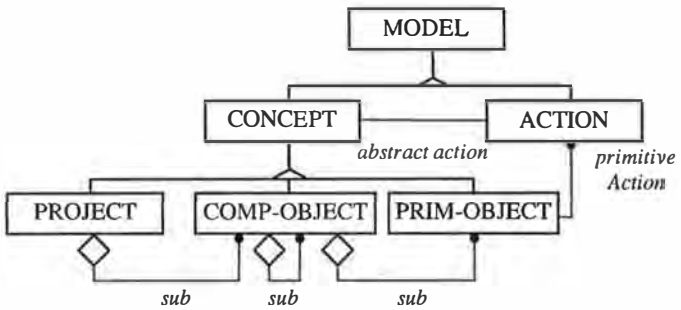


Figure 7-6, Domain model pattern

Class *MODEL* is *abstract*, and will therefore never be instantiated. The class serves two purposes. Primarily, it types all classes used to model a domain as being of type *MODEL*. It is therefore possible to write operators which work over the whole model. Secondly, attributes and operations common to all model elements may be specified in one location.

The abstract class *CONCEPT* supports the operations and attributes common to all classes which may be used to model the objects within a domain. The class contains an attribute permitting actions to be attached to objects through the abstract action relationship.

Classes *PROJECT*, *COMPOSITE-OBJECT*, and *PRIMITIVE-OBJECT* may be instantiated. Class *PROJECT* makes explicit a special type of class within MBP which defines the whole project to be constructed. This class may not be the subcomponent of any other class. Class *COMPOSITE-OBJECT* may be decomposed into other instances of class *COMPOSITE-OBJECT* and class *PRIMITIVE-OBJECT*. Instances of class *PRIMITIVE-OBJECT* may not have subcomponents.

Each of the classes within the model is defined in a table below.

Abstract Class *MODEL*		
Description	Parent of all model elements. The class defines each model element as being of type model and supports the implementation constructs for automatically generating instance names.	
Attribute	**Init. Val**	**Description**
Default Name	user define	Default name prefix for instances of a class
Naming Convention	auto | user	Indicates that when an instance is required if the system should define the name or if the user should be consulted
Next Instance Number	0	Concatenated with default name to create a unique name.
Method	**Param.**	**Description**
Generate Name	none	If naming convention is set to *auto* the method combines the *default name* attribute and the *next instance number* attribute to generate a new and unique name. If naming convention is set to *user*, the user is prompted for the name of an instance.

Table 1, Class *MODEL*

Abstract Class *CONCEPT*, subclass of *MODEL*		
Description	Provides the constructs general to all classes which model objects in a domain.	
Attribute	**Init. Val**	**Description**
Abstract action	user define	Action which describes the artefact
Method	**Param**	**Description**
[Generate Dependency]	none	Deferred method which must be written by the user for each class to determine dependency between actions
Constraints		
One action only may be related through the abstract action relation		

Table 2, Class *CONCEPT*

Class *PROJECT*, subclass of class *CONCEPT*		
Description	The instance of this class represents the specific problem within an application domain for which a plan is to be synthesised.	
Attribute	**Init. Val**	**Description**
Sub	user	List of sub concepts
Constraints		
All subcomponents of the project must be reachable from this class via the Sub relation. A model must contain one and only one instance of class project		

Table 3, Class *PROJECT*

Class *COMPOSITE-OBJECT*, subclass of class *CONCEPT*		
Description	Objects which may be decomposed into other objects	
Attribute	**Init. Val**	**Description**
Super	user	list of concepts of which this class is a sub concept
Sub	user	sub concepts of this class
Constraints		
Instance of this class may not be related to an instance of class project through the sub relation. No object related as an sub concept may be related as a super concept of this class.		

Table 4, Class *COMPOSITE-OBJECT*

Class *PRIMITIVE-OBJECT* subclass of *CONCEPT*		
Description	Defines objects which may not have subcomponents	
Attribute	**Init. Val**	**Description**
Primitive action	user	List of primitive actions associated with this concept
Super	user	List of the super concepts of this concept.

Table 5, Class *PRIMITIVE-OBJECT*

Class *ACTIONS*, subclass of *MODEL*		
Description	Describes an action within a domain	
Attribute	**Init. Val**	**Description**
Object	system	Defined by the system when an instance of class action is attached to a concept
Main effects	user	The effects which make up the purpose of this action
Side effects	user	The effects for which this action would not be used, but do occur as a result of applying this action.
Preconditions	user	List of aggregate conditions which must be met before this action may be applied.
Achieve conditions	user	List of conditions for the HTN planning engine to make true.
Sub	system	List of actions which are sub may be viewed as sub concepts of this action
Super	system	List of actions of which this action is a sub action

Table 6, Class *ACTION*

7.3.4 Domain specific relationships

Subclasses of *PROJECT, COMPOSITE-OBJECT*, and *PRIMITIVE-OBJECT* may have domain specific relationships attached. The pattern for specifying such relationships is depicted in Figure 7-7 below.

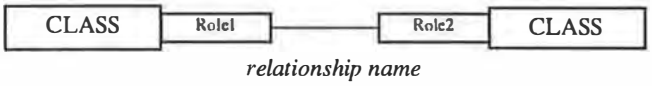


Figure 7-7, Generic relationship template

To create the *supported* relationship, the domain writer must take the following steps. First, identify the two roles within the relationship. In the case of *support,* the roles are *supportee* and *supporter*. The class participating as the *supportee* will require an attribute defined *supported by*, whilst the *supporter* class will have the attribute *supports.*

7.3.5 Facets

Each attribute within a class is composed of three facets[1]:

- **Value**: The actual value of the attribute
- **Default**: A default value for the attribute
- **If_Needed**: Specifies the inference mechanisms which must be invoked to infer the value of the attribute.

Attributes are queried with two types of request: *return (attribute, specific facet)* or *return (attribute).* The specific query returns the value in the facet requested, or *unknown* if the facet is undefined. In the case of the *if_needed* facet, the appropriate inference mechanism is invoked to derive the value (see section 7.3.6 for a detailed description). The second query searches the three facets of an attribute in a defined order until a value can be obtained or inferred. The value facet is examined first and the value returned if defined. If the value facet is undefined, the default facet is examined and returned if defined. If the value and the default facets are undefined, the if_needed facet inference is activated to derive the attributes value.

When activated, the if_needed facet updates the value facet of an attribute under the rationale that the most recently inferred value is the most accurate.

[1]This modelling is based upon the RBFS (Rule Based Frame System) (Barber, Marshall, & Boardman 1987)

7.3.6 Instantiation directives and inference packages

Collectively *instantiation directives* and *inference packages* provide the mechanism for attaching inference knowledge to the *if_needed* facet of an attribute. *Inference packages* encapsulate reasoning mechanisms, whilst *instantiation directives* both define the interface to inference packages and the how the results of the package will be processed. The MBP provides a set of predefined instantiation directives. It is the responsibility of the domain writer to provide inference packages. The domain writer may add new instantiation directives; however, modifications of this type are considered changes to the MBP inference engine. The writing of instantiation directives is therefore described within the description of the inference engine (section 7.4.1.2).

The following sub sections define the set of instantiation directives supported by the model-based planner and the constraints each directive places upon the inference packages it may invoke and process. Each instantiation directive may be attached only to specified class attribute pairs. Hence, the presentations of directives below groups the directives under the heading of the class attribute pairs to which they may be applied.

7.3.6.1 *CONCEPT*.abstract-actions and *PRIMITIVE-OBJECT*.primitive-actions

Must (<?cardinality, ?action-class-name>)

Directive *must* specifies that a number of instances of the class defined by the *?action-class-name* parameter must be created and linked to the instance to which the directive is attached. The number of instances of class *?action-class-name* generated is dependent upon the *?cardinality* parameter which may range from *a* (one) to *infinity*[2]. The following model fragment illustrates the use of the must directive.

```
class BEAM
    abstract action :  value        = ∅
                       default      = ∅
                       if_needed    = must(a, LAY)
beam1: BEAM
```

The query *return-value (beam1, abstract action)* will result in one instance of the class *LAY* being instantiated, and the value facet of *beam1* being updated to equal the name of that instance.

[2] As a MBP is implemented on a physical and therefore finite machine, the number of instances permitted is limited by the physical resources of that machine. The notion of a plan containing an infinite number of steps leads to some interesting issues - particularly execution time.

Infer (<?cardinality, ?action-class-name, ?inference-package, ?optional-parameters>)

Directive *infer* specifies that a number of instances of the class denoted by the *?action-class-name* parameter may be created and linked to the instance of the class to which the directive is attached. The criteria for determining if the instances are or are not required is the result of the *?inference-package* parameter. The *infer* directive constrains its associated inference packages to return *true* or *false*. Where *true* indicates that the action class should be associated. The following model fragment illustrates the use of the *infer* directive and the inference package mechanism which must be attached.

```
class BEAM
  primitive-action :value = ∅
                    default = ∅
                    if_needed = infer (a, STRIKE-FORMWORK,
                    infer_strike_formwork, null)
beam1: BEAM
```

```
inference package infer-strike-formwork
  goal strike-formwork
  backward-chain
  rule1 cost 0
      if beam.formwork-type = custom then
          return true
  end rule1

  rule2 cost 0
      if beam.formwork-type = prefabricated then
          return false
  end rule2
```

The query *return value (beam1, primitive-action)* will result in one instance of class *strike formwork* being instantiated and linked to instance beam1 if an only if *infer_strike_formwork* returns *true*.

7.3.6.2 *ACTION*.main-effects and *ACTION*.side-effects

action-must (?effect-text)

Directive *action-must* specifies that the parameter *?effect-text* must be added to the effect slot of any instance of the class to which it is attached. The *?effect-text* parameter may contain a number of variables which the *action-must* directives instantiates. These variables are detailed below:

- *?object* is instantiated to the name of the object instance to which the action is associated.
- *?attribute-name* is instantiated the value of the attribute corresponding to *?attribute-name* on the object instance to which the action is associated.
- *?relationship-name* is instantiated to the name of the object instance to which the object the action is associated with is related through *?relationship-name.* This parameter may only be used if the relationship is constrained to be one to one.

The example below demonstrates the use of this directive. Examples of the variable instantiations may be found within the Pacifica domain encoding in Chapter 9.

```
class LAY
    related-object:     value =∅
    main-effects:value = ∅
                        default = ∅
                        If_needed = action-must(laid ?object = true)
lay1: LAY
    related-object:     value = beam1
```

The query *return (lay1, main-effects)* will result the value facet of *lay1's main-effect* attribute being set to *laid beam1 = true.*

action-infer (?effect-text, ?inference-package, ?optional-parameters)

Directive *action-infer* specifies that the parameter *?effect-text* may be added to the effect slot of the instance of the class to which it is attached. Determining if an effect is or is not required is achieved through the inference package identified by the *?inference-package* parameter. The directive constrains inference packages to return *true* and *false*. The *?effect-text* parameter is associated if and only if the inference package returns *true*. The example below illustrators the use of this directive.

```
class LAY
    related-object:    value =∅
    main-effects:value = ∅
                       default = ∅
                       if_needed = action-infer (vibration at ?object = high,
                       infer_vibration_effect)
lay1: LAY
    related-object:    value = pile1
```

The query *return (lay1, main-effects)* will result in the value facet of *lay1's main-effects* attribute being set to *vibration at pile1 = high* if and only if the inference package *infer_vibration_effect* determines that the laying of *pile1* will cause excess vibration.

7.3.6.3 ACTION.preconditions

aggregate-condition (?condition)

The *Aggregate condition* directive updates the value facet of the attribute to which it is attached with a list of conditions which must hold before the single *?condition* parameter may be considered satisfied. For example, the condition *dry building* will require a set of conditions to hold. Membership of this set is dependent upon the components in specific instance of a problem. If a building includes a large concrete floor, the effect *floor = dry* will form part of the *dry building* condition. If the building does not contain a large concrete floor then this effect is not required.

The domain writer must provide a method which returns the list of conditions which together constitute the aggregate condition.

7.3.7 Dependency assessment

Each subclass of class *CONCEPT* must have a *determine dependency* method written. The method may contain two types of dependency assessment knowledge: relationship dependency and, in the case of primitive objects only, primitive dependency. The relationship dependency mechanism determines the ordering constraints which must be added between actions as a result of the objects to which they are related through a domain specific relationship. The primitive dependency mechanism determines the ordering constraints which should be added between a primitive object's primitive action set.

In the context of a MBP domain model, a method is an unit of procedural code which is executed sequentially. The domain writer may therefore write procedural code to determine the dependency between actions or access other forms of inference mechanism. A number of dependency directives are supplied which the domain writer may insert into a determine dependency method. Each directive is described below.

Link-abstract-action-with-main-effects (?relationship)

Let A be the instance to which the directive has been attached. Invoking the directive will identify the set B which is composed of the objects related to A by the relationship parameter *?relationship.* The directive then generates the set X composed of the abstract actions of set B. The abstract action of A is then made dependent upon the actions in set X. The *main-effects* of set X are appended to the *precondition* attribute of the abstract action of A.

Link-abstract-action-without-effects (relationship)

This directive performs the same operation as the directive *link-abstract-action-with-main-effects (?relationship)* with the exception of copying the effects of set X to the abstract action of A. The directive does not copy effects.

Infer-link-abstract-action-with-main-effects(?relationship, ?inference-package)

This directive performs the same operation as the directive *link-abstract-action-with-main-effects (?relationship)* with the exception of determining the membership of set B. The directive applies the inference package specified by the parameter *?inference-package* to determine if each of the objects related to A by the relationship parameter *?relationship* should be included within set B.

Infer-link-abstract-action-without-effects (?relationship, ?inference package)

This directive performs the same operation as the directive *infer-link-abstract-action-with-main-effects* with the exception of copying the effects of set X to the abstract action of A. The directive does not copy effects.

Primitive-dependency (?primitive-action-head, ?primitive-action-tail)

This directive may be applied only to a primitive objects primitive action attribute. The directive adds a dependency relationship *{head < tail}*. Where *head* corresponds to the parameter *?primitive-action-head* and *tail* to the parameter *?primitive-action-tail.*

Infer-primitive-dependency (?primitive-action-head, ?primitive-action-tail, ?inference-package)

This directive may be applied only to a primitive objects primitive action attribute. The directive may add a dependency relationship *{head < tail}*. Where *head* corresponds to the parameter *?primitive-action-head* and *tail* to the parameter *?primitive-action-tail.* A relationship is added if and only if the inference package parameter *?inference-package* returns true.

7.3.8 Encoding an example domain

This section demonstrates application of the domain modelling constructs described above through encoding an example problem from the construction industry.

7.3.8.1 Step 1: encode the KADS model of expertise

The KADS model of expertise, detailed within Chapter 5, identified a number of classes within the construction domain related through the aggregation relationship. Each class in the KADS model may be translated into a class in the model-based representation. Aggregation relationships are translated into sub and super component attributes of classes as appropriate.

```
class BUILDING
  subclass of PROJECT
  sub         {FOUNDATIONS, ROOF}
class ROOF
  subclass of COMPOSITE-OBJECT
  super       BUILDING
  sub         {STEELWORK, ROOF-DECK, ROOF-COVERING}
class FOUNDATIONS
  subclass of COMPOSITE-OBJECT
  super       BUILDING
  sub         {BEAM, PILE}
class STEELWORK
  subclass of PRIMITIVE-OBJECT
  super       ROOF
class ROOF-DECK
  subclass of PRIMITIVE-OBJECT
  super       ROOF
class ROOF-COVERING
  subclass of PRIMITIVE-OBJECT
  super       ROOF
class BEAM
  subclass of PRIMITIVE-OBJECT
  super       FOUNDATIONS
  formwork type   prefabricated | custom
class PILE
  subclass of PRIMITIVE OBJECT
  super       FOUNDATIONS
```

Figure 7-8, Translation from KADS model to MBP classes

7.3.8.2 Step 2: attach relationships

The second phase of the domain modelling captures domain specific relationships. Each relationship in the KADS domain model is translated into an attribute on the classes which may participate in the relationship. Figure 7-9 below develops Figure 7-8 to include domain specific relationships (bolded).

```
class BUILDING
      subclass of   PROJECT
      sub           {FOUNDATIONS, ROOF}
class ROOF
      subclass of   COMPOSITE OBJECTS
      super         BUILDING
      sub           {STEELWORK, ROOF-DECK, ROOF-COVERING}
class FOUNDATIONS
      subclass of   COMPOSITE-OBJECTS
      super         BUILDING
      sub           {BEAM, PILE}
class STEELWORK
      subclass of   PRIMITIVE-OBJECT
      super         ROOF
      supports      {ROOF-DECK}
      supported by  {BEAM}
class ROOF-DECK
      subclass of   PRIMITIVE-OBJECT
      super         ROOF
      supports      {ROOF-COVERING}
      supported by  {STEELWORK}
class ROOF-COVERING
      subclass of   PRIMITIVE-OBJECT
      super         ROOF
      supported by  {ROOF-DECK}
class BEAM
      subclass of   PRIMITIVE-OBJECT
      super         FOUNDATIONS
      formwork type prefabricated | custom
      supported by  {PILE}
      supports      {STEELWORK}
class PILE
      subclass of   PRIMITIVE-OBJECT
      super         FOUNDATIONS
      supports      {BEAM}
```

Figure 7-9, MBP classes with domain specific relationships encoded

7.3.8.3 Step 3 : attach abstract and primitive actions

Step 3 attaches abstract and primitive actions to each subclass of concept within the model. Figure 7-10 below extends Figure 7-8 to include abstract and primitive actions.

class BUILDING	
subclass of	PROJECT
sub	{FOUNDATIONS, ROOF}
abstract action:	**if_needed must(a, build)**
class ROOF	
subclass of	COMPOSITE-OBJECTS
super	BUILDING
sub	{STEELWORK, ROOF DECK, ROOF COVERING}
abstract action:	**if_needed must(a, erect)**
class FOUNDATIONS	
subclass of	COMPOSITE-OBJECTS
super	BUILDING
sub	{BEAM, PILE}
abstract action:	**if_needed must(a, lay)**
class STEELWORK	
subclass of	PRIMITIVE-OBJECT
super	ROOF
supports	{ROOF-DECK}
abstract action:	**if_needed must(a, erect)**
primitive actions:	**if_needed must(a, primitive erect)**
class ROOF-DECK	
subclass of	PRIMITIVE-OBJECT
super	ROOF
supports	{ROOF-COVERING}
supported by	{STEELWORK}
abstract action:	**if_needed must(a, lay)**
primitive actions:	**if_needed must(a, primitive lay)**
class ROOF-COVERING	
subclass of	PRIMITIVE-OBJECT
super	ROOF
supported by	{ROOF-DECK}
abstract action:	**if_needed must(a, lay)**
primitive actions:	**if_needed must(a, primitive lay)**
class BEAM	
subclass of	PRIMITIVE-OBJECT
super	FOUNDATION
formwork type	prefabricated | custom
supported by	{PILE}
abstract action:	**if_needed must(a, lay)**
primitive action:	**if_needed must(a, setoutposition) must(a, excavate beam) must(a, blind bottom) must(a, reinforcement cages) must (a, formwork) must(a, clean out) must(a, concrete) must(a, cure concrete) must(a, vibrate concrete) infer(a, mould oil, DETERMINE MOULD OIL) infer(a, strike formwork, DETERMINE STRIKE FORMWORK)**
class PILE	
subclass of	PRIMITIVE-OBJECT
sub	FOUNDATION
supports	{BEAM}
abstract action:	**if_needed must(a, lay)**
primitive actions:	**if_needed must(a, lay pile mat), must(a, drive pile), must(a, prepare ground)**

Figure 7-10, Attachment of abstract and primitive actions to classes in the MBP domain model

7.3.8.4 Step 4: write action classes

An action class must be written for each action which may be associated with a subclass of class *CONCEPT*. Figure 7-11 defines the action classes required by Figure 7-10.

```
class BUILD
  subclass of      ACTIONS
  object           { }
  main effects     action must(?object = built)
  side effects     { }
  achieve          { }
  preconditions    { }
class BUILD-BUILDING
  subclass of      BUILD
  object           {BUILDING}
class ERECT
  subclass of      ACTIONS
  object           { }
  main effects     action must(?object = erected)
  side effects     { }
  achieve          { }
  preconditions    { }
class ERECT-ROOF
  subclass of      ERECT
  object           {ROOF}
class ERECT-STEELWORK
  subclass of      ERECT
  object           {STEELWORK}
class LAY
  subclass of      ACTIONS
  object           { }
  main effects     action must(?object = laid)
  side effects     { }
  achieve          { }
  preconditions    { }
class LAY-FOUNDATIONS
  subclass of      LAY
  object           {FOUNDATIONS}
  precondition     aggregate condition {site-safe}
class LAY-ROOF-DECK
  subclass of      LAY
  object           {ROOF-DECK}
class LAY-ROOF-COVERING
  subclass of      LAY
  object           {ROOF-COVERING}
class LAY-PILE
  subclass of      LAY
  object           {PILE}
class LAY-BEAM
  subclass of      BEAM
  object           {BEAM}
class SET-OUT-POSITION
  subclass of      ACTION
  main effects     action must (?object position = set out)
```

Figure 7-11, Action classes

7.3.8.5 Step 5: write inference packages

Figure 7-12 defines the inference packages to determine the need for the actions from the directive *infer* in Figure 7-10.

```
inference package-determine-mould-oil
  goal mould oil
  backward chain
  rule 1
  if formwork type of beam = custom then
       true
  end rule
  rule 2
  if formwork type of beam = prefabricated then
       false
  end rule

inference package determine-formwork
  goal formwork
  backward chain
  rule 1
  if formwork type of beam = custom then
       true
  end rule
  rule 2
  if formwork type of beam = prefabricated then
       false
  end rule
```

Figure 7-12, Inference packages

7.3.8.6 Step 6: write methods for determining dependency

Each subclass of concept requires a determine dependency method to be written. Figure 7-13 specifies the determine dependency methods required by the construction domain model.

```
class BUILDING
    method determine-dependency
        {
            null
        }

class ROOF-DECK
    supports        {ROOF-COVERING}
    supported by {STEELWORK}
    method determine dependency
        {
            link-abstract-action-with-main-effects (supported_by)
        }

class ROOF-COVERING
    supported by {ROOF-DECK}
    method determine dependency
        {
            link-abstract-action-with-main-effects (supported_by)
        }

class BEAM
    formwork type    prefabricated | custom
    supported by {PILE}
    method determine dependency
        {
            link-abstract-action-with-main-effects (supported-by)
            primitive-dependency (SET-OUT-POSITION, PREPARE-GROUND)
            primitive-dependency (PREPARE-GROUND, EXCAVATE-BEAM)
            primitive-dependency (EXCAVATE-BEAM, BLIND-BOTTOM)
            primitive-dependency (BLIND-BOTTOM, FORMWORK)
            primitive-dependency (FORMWORK, CLEAN-OUT)
            primitive-dependency (MOULD-OIL, CLEAN-OUT)
            primitive-dependency (CLEAN-OUT, REINFORCEMENT-CAGES)
        }

class PILE
    supports        {BEAM}
    method determine dependency
        {
            primitive-dependency (PREPARE-GROUND, LAY-PILE-MAT)
            primitive-dependency (LAY-PILE-MAT, DRIVE-PILE)
        }
```

Figure 7-13, Methods for determining dependency

7.3.8.7 Step 7: add instances for a specific problem

Steps one to six defined the generic action and dependency knowledge of the construction problem elicited in Chapter 5. Step 7 instantiates all subclasses of class *CONCEPT* to represent specific instances and relationships in a building's design.

instance supermarket	
instance of class	BUILDING
sub	{the-foundations, the-roof }
instance the roof	
instance of class	ROOF
sub	supermarket
sub	{roof-steelwork, the-roof-deck, the-roof-covering}
instance foundations	
instance of class	FOUNDATIONS
sub	supermarket
sub	{beam1, pile1, pile2}
instance roof steelwork	
instance of class	STEELWORK
sub	the-roof
supports	{the roof deck}
instance the roof deck	
instance of class	ROOF-DECK
sub	the-roof
supports	{the-roof-covering}
supported by	{roof-steelwork}
instance the roof covering	
instance of class	ROOF-COVERING
sub	the-roof
supported by	{the-roof-deck}
instance beam1	
instance of class	BEAM
sub	the-foundation
formwork type	custom
supported by	{pile1, pile2}
instance pile1	
instance of class	PILE
sub	the-foundation
supports	{beam1}
instance pile2	
instance of	class PILE
sub	the foundation
supports	{beam1}

Figure 7-14, Instances representing the restaurant extension

7.4 Model-based planner

The model-based planner component processes a domain model specified using the constructs in section 7.3 to synthesise actions, conditions, effects, and dependency constraints.

The MBP component is divided into two sets of algorithms: planner support functions, and planning algorithms. The support functions process the facets of attributes invoking and processing instantiation directives and inference packages. The planning algorithms navigate a domain model, utilising the planner support functions, to generate the actions, conditions, effects, and dependency constraints required.

Planner support functions are described before the planning functions below as the planning functions are implemented in terms of the support functions. After both sets of functions have been introduced, an example execution over the domain model specified in section 7.3 is described.

7.4.1 Planner support functions

Planner support functions may be decomposed into two sets: facet functions, and instantiation directive and inference package handlers. Each set is described below.

7.4.1.1 Facet functions

Facets functions process the three facet attribute structure. The following table describes the functions, their parameters, and their operation.[3]

Function	parameters	description
o_get	instance, attribute, facet.	returns the value or inferred value of a specific instance's attribute facet.
o_get_v_d_f	instance, attribute.	following the precedence value, default, if_needed, this function returns the value of an attribute.
o_put	instance, attribute, facet, value.	updates the specific facet of an attribute to include the value parameter.
o_clear	instance, attribute, facet.	clears all data from the specified instance, attribute, facet triple.

[3]These functions are based upon the Rule-Based Frame System (RBFS) (Barber et. al. 1987). Function prefixes have been changed from *f_* to *o_* to reflect the use of object-oriented technology within the domain modelling.

A fragment of pseudo code demonstrating the processing of an *if_needed* facet within both *o_get* and *o_get_v_d_f* is depicted below:

```
if facet = if_needed then

  Identify directive
  Result = Invoke directive
  if result not equal to unknown or result equal to already-processed then
      o_put(self, attribute, value, result)
  end if
  return result.
```

Figure 7-15, if_needed facet processing pseudo code

It is important to note that querying the *if_needed* facet of an attribute will update the *value* facet if a result is obtained. The inferred value will be the most up to date value for an attribute, thus, it should over write any the existing *value* facet. The exception to this case is if an instantiation directive returns *already-processed*. This result permits the domain writer the option the to write instantiation directives which instantiate and update the *value* facet of an attribute if the default mechanism is not appropriate. The domain writer must, however, ensure that the *value* facet of an attribute is updated to equal the most recently referred value.

7.4.1.2 Processing instantiation directives and inference packages

The processing of instantiation directives and inference packages is initiated by either an *o_get_v_d_f* or an *o_get* request for an attribute's *if_needed* facet. Figure 7-15 presents the pseudo code for the directive handler module of the MBP. Figure 7-16, below, places this algorithm into context with the instantiation directive and inference package components of the MBP.

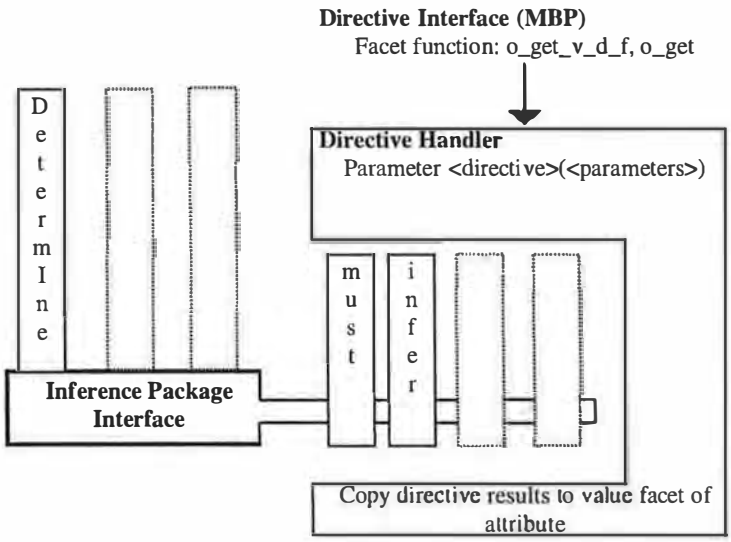


Figure 7-16, Instantiation and inference package processing architecture

The directive interface consists of the two facet functions *o_get* and *o_get_v_d_f*. If either function is invoked for an *if_needed* facet of an attribute, the directive handler module is invoked. All directives conform to the syntax *<directive>(<parameters>)*. The directive handler identifies the specific directive stored in an attribute and invokes that directive. In Figure 7-16, directives are represented as rectangular sub modules of the directive handler model. The directives *must* and *infer* are included in the figure together with two dotted rectangles - indicating that more directives may be added to the architecture.

New directives may be supplied by a domain writer as both part of the MBP and as domain specific directives stored on the class which requires them. This architecture allows the domain writer to identify new domain independent directives and integrate them into the MBP, or to encode domain specific directives within the domain model. Domain specific and domain independent constructs are thus kept separate. Figure 7-17, below, presents the directive search routine which first searches the MBP for an instantiation directive, before searching the class on which the attribute to which the *if_needed* facet is attached.

```
if method on MBP
    call directive handler
if method on calling class
    call directive handler
else
    error, undefined directive
end if
```

Figure 7-17, Directive search routine

During processing, instantiation directives may utilise the *inference package interface*, depicted in Figure 7-16 as resting behind the instantiation directives. The inference package interface allows instantiation directives to invoke inference packages written for a specific domain.

7.4.2 Planning functions

Planning functions apply the planner support functions described in section 7.4.1 to synthesise the actions and dependency constraints required by a plan. Each planning function is introduced below, before the overall planning algorithm is described.

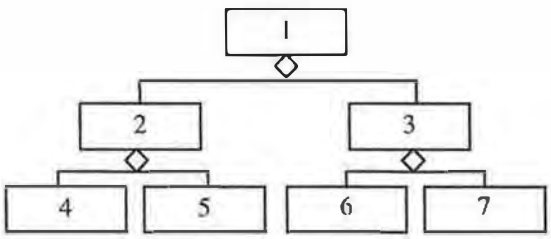
7.4.2.1 Action synthesis

Each instance of a subclass of class *CONCEPT* within a domain model will have an abstract action attribute and instances of class *PRIMITIVE-OBJECT* will have in addition a primitive action attribute. By querying the value of these attributes using *o_get_v_d_f*, the action instances which need to be associated with each instance in a domain model will be synthesised via instantiation directives and inference packages. The action synthesis algorithm applies the *o_get_v_d_f* query to each primitive and abstract action attribute within a domain model. Figure 7-18, below, presents the action synthesis algorithm.

```
generate actions (item: COMPONENT)
  o_get_v_d_f(item, abstract-action)
  for each action:   o_get_v_d_f(action, main-effects)
                     o_get_v_d_f(action, side-effects)
  if item is of type PRIMITIVE-OBJECT then
      o_get_v_d_f(item, primitive-actions)
      foreach action:   o_get_v_d_f(action, main-effects)
                        o_get_v_d_f(action, side-effects)
  else
      for each sub item, n or item
          generate actions (n)
  end if
```

Figure 7-18, Action synthesis algorithm

The algorithm is initiated with the name of the single instance of class *PROJECT* within a domain model. The abstract action attribute is queried, before the algorithm is applied recursively to each instance in the domain model related through the *sub* relation with the project instance. The path taken through a domain model by the algorithm is depicted in Figure 7-19 below.



Path = 1,2,4,5,3,6,7

Figure 7-19, Action synthesis model transversal order

The order of model traversal in not significant at present. The only constraint upon the algorithm is that every instance of a subclass of concept is processed by the algorithm. Issues surrounding the order of model transversal are discussed in the further work section within Chapter 9.

7.4.2.2 Updating the hierarchy of actions

The action synthesis algorithm results in a number of model elements associated with actions. The action hierarchy update algorithm processes each of the actions within a model to add *sub* and *super* relationships. Abstract actions are linked with their related objects super components abstract action under the *super* relationship. Primitive actions are related to their related components abstract action through the *super* component relationship. Figure 7-20, below, presents the input and output of this process.

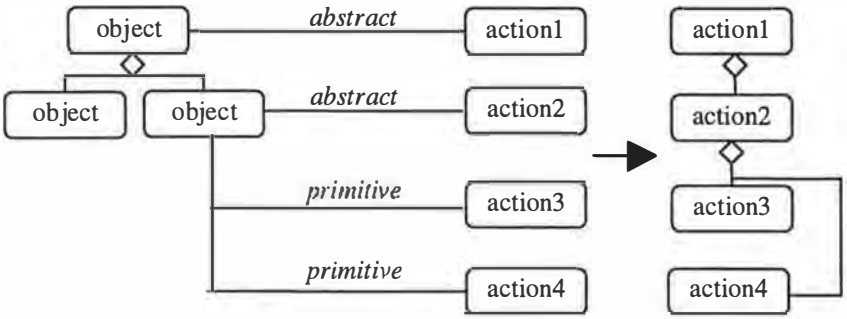


Figure 7-20, Action hierarchy

The action hierarchy update algorithm is detailed in Figure 7-21 below.

```
update-hierarchy (object: CONCEPT)
for all actions

  if action.object is not of type PROJECT and is not of type PRIMITIVE-OBJECT then
          action.sub = action.object. super.abstract-action
          action.object. sub.abstract-action = action.sub
  if action is a primitive action then
          action.sub =action.object. abstract
          action.object. abstract.sub = action
  end if
```

Figure 7-21, Action hierarchy update algorithm

7.4.2.3 Activity generation

MBP defines an activity as the union of an object and an action. The activity generation algorithm recurses each action within a model and generates an activity which records the name of the action and the object associated with it. Figure 7-22, below, presents the activity generation algorithm.

```
generate-activities (object: COMPONENT)
for all actions
        create a new instance of class activity
        new activity.action = action
        new activity.object = action. object
```

Figure 7-22, Activity synthesis algorithm

The activity data structures resulting from this process are utilised by the dependency synthesis algorithms and the HTN planner interface described in sections 7.4.2.4 and 7.5 respectively.

7.4.2.4 Dependency synthesis

Dependency synthesis derives the dependency constraints between activities through two stages: *relationship and primitive dependency* and *aggregate dependency*. Each process is described below.

7.4.2.4.1 Relationship and primitive dependency

Each subclass of class *CONCEPT* must have a method *determine dependency* written. The relationship and primitive dependency algorithm traverses the model activating this method for each instance. The dependency synthesis algorithm is defined in Figure 7-23 below

```
generate dependency over model (Item : component)
  send message(item : generate-dependency)
  if item is not primitive then
            for each sub item, n
                 generate-dependency (n)
            end for
  end if
```

Figure 7-23, Dependency synthesis algorithm

7.4.2.4.2 Aggregate dependency

Aggregate dependency is invoked by applying the *o_get_v_d_f* query to each instance of class *ACTION* within the domain model. This algorithm is specified below.

```
for all instances of class ACTION
        o_get_v_d_f (action, precondition)
```

7.4.2.5 Complete MBP planning algorithm

The MBP algorithm applies the activity synthesis, action hierarchy update, activity synthesis and dependency synthesis algorithms defined above.

```
MBP_plan(project : PROJECT)
        generate-object-actions (project)
        update-hierarchy(project)
        generate-activities(project)
        generate-dependency(project)
```

Figure 7-24, Complete MBP algorithm

7.4.3 Applying the MBP algorithm

This section presents the execution trace of the MBP algorithms described above produced when synthesising a plan for the domain description produced in section 7.3.8. The text has been formatted for clarity.

```
user >Invoke MBP
---- MBP v4 invoked ---
one instance of class project = supermarket
planning for supermarket
generate actions (supermarket)
    o_get_v_d_f(supermarket.abstract-action) returns build
        :because: must(a, build) directive
    supermarket is not primitive
    processing subcomponents of supermarket
    for all subcomponents of supermarket over set = {the-foundations, the-roof}
        looping for the-foundations
        generate-actions (the-foundations)
            o_get_v_d_f(the-foundation.abstract-action) returns lay
            :because of must(a, lay) directive
            the-foundations is not primitive
            for all subcomponents of the-foundations over set = {beam1, pile1, pile2}
            looping for beam1
            o_get_v_d_f(beam1.abstract-action) return lay
            :because of must(a lay) directive
            beam1 is a primitive-object
            o_get_v_d_f(beam1.primitive-action) returns
            {set-out-position, excavate-beam, blind-bottom, lay-reinforcement-
            cages, formwork, clean-out, cure-concrete, vibrate-concrete}
            :because of must directive:
            {mould-oil}
             because of infer directive and inference package
            DETERMINE MOULD OIL returned true
            {strike-formwork}
            : because of infer directive and inference package
            DETERMINE STRIKE FORMWORK returned true
            processing of beam1 complete
            looping for pile 1
                o_get_v_d_f(pile1.abstract-action) return lay
                :because of directive must(a, lay)
                pile1 is a primitive object
                o_get_v_d_f(pile1.primitive-actions) returns
                {lay-pile-mat, drive-pile, prepare-ground}
                :because of directive must:
                processing of pile1 complete
            looping for pile2
                o_get_v_d_f(pile2.abstract-action) return lay
                :because of directive must(a, lay)
                pile2 is a primitive-object
                o_get_v_d_f(pile2.primitive-actions) returns
                {lay-pile-mat, drive-pile, prepare-ground}
                :because of directive must:
                processing of pile2 complete
                processing of the-foundations complete
        looping for the roof
            generate actions(the-roof)
                o_get_v_d_f(the-roof.abstract-action) return erect
                :because of directive must(a, erect)
                the roof is not primitive
                for all subcomponents of the-roof over set = {the-roof-deck, the-
roof-steelwork, the-roof-covering}
                    looping for the-roof-deck
            o_get_v_d_f(the-roof-deck, abstract-action) return lay
        : because of direct must (a, lay)
        the-roof-deck is primitive
        o_get_v_d_f(the-roof-deck, primitive-action) return primitive lay
        :because of directive must(a, primitive lay)
```

```
        processing of the roof-deck-complete
        looping for the roof-steelwork
        o_get_v_d_f(the-roof-steelwork, abstract-action) return erect
        : because of direct must (a, erect)
        the-roof-steelwork is primitive
        o_get_v_d_f(the-roof-steelwork, primitive-action) return primitive-erect
        :because of directive must(a, primitive-lay)
        processing of the-roof-steelwork complete
        looping for the-roof-covering
        o_get_v_d_f(the-roof-covering, abstract-action) return lay
        : because of direct must (a, lay)
        the-roof-steelwork is primitive
        o_get_v_d_f(the-roof-steelwork, primitive-action) return primitive lay
        :because of directive must(a, primitive lay)
        processing of the-roof-covering complete
        finished subcomponents of the-roof
        finished processing of the-roof
        finished processing of the-supermarket
    generation of actions completed
```

Action synthesis results in a set of actions associated with the instances of class concept in a model. These instances are detailed below. Modifications to the instances by the action synthesis algorithm are **bolded**.

```
---- MBP Instances of class concepts list
instance supermarket
        instance of class     BUILDING
        sub                   {the foundations, the roof}
        abstract action:      value a build action 1
                              if_needed must(a, build)
instance the-roof
        instance of class     ROOF
        super                 supermarket
        sub                   {roof steelwork, the roof deck, the roof covering}
        abstract action       value a erect action 1
                              if_needed   must(a, erect)
instance the-foundations
        instance of class     FOUNDATIONS
        super                 supermarket
        sub                   {beam1, pile1, pile2}
        abstract action:      value a lay action1
                              if_needed: must(a, lay)
instance steelwork
        instance of class     STEELWORK
        super                 the roof
        supports              {the roof deck}
        abstract action       value an erect action 1
                              if_needed must(a, erect)
        primitive actions     value a primitive erect action 1
                              if_needed must(a, primitive erect)
instance beam1
        instance of class     BEAM
        super                 the-foundations
        formwork type         custom
        supported by          {pile}
        abstract action       value a lay action 2
                              if_needed must(a, lay)
        primitive action      value  a set out position action 1, a excavate beam action 1
                              a blind bottom action 1, a reinforcement cage action 1
                              a formwork action 1, a clean out action 1, a concrete action 1
                              a cure concrete action1, a vibrate concrete action 1
                              a mould oil action 1, a strike formwork action 1
                              if_needed must(a, setoutposition) must(a, excavate beam)
                              must(a, blind bottom) must(a, reinforcement cages)
                              must (a, formwork) must(a, clean out)
                              must(a, concrete) must(a, cure concrete)
                              must(a, vibrate concrete)
                              infer(a, mould oil, DETERMINE MOULD OIL)
```

infer(a, strike formwork, DETERMINE STRIKE FORMWORK)

The attachment of actions to subclasses of class concept results in the instantiation of new actions. A subset of the new actions is listed below. The object to which each action instance is related is **bolded**.

```
--- MBP action instances

instance lay1
        instance of class lay
        object    the foundations
        main effects the foundations laid = true
        sub       { }
        super     { }

instance lay2
        instance of class lay
        object    pile1
        main effects pile1 = laid
        sub       { }
        super     { }

instance lay pile mat1
        instance of class lay pile mat
        object    pile 1
        main effects pile mat = laid
        sub       { }
        super     { }
instance drive pile
        instance of class drive
        object    pile 1
        main effect pile1 = driven
        sub       { }
        super     { }

instance prepare ground
        instance of class prepare ground
        object    pile 1
        main effect ground pile1 = prepared
        sub       { }
        super     { }
--- MBP end of action instance trance
```

With action synthesis complete, the MBP algorithm invokes the *update action hierarchy* algorithm to set the *sub* and *super* relationships of each action. A trace of this process is presented below. The updated actions are output immediately after the algorithm trace.

```
--- MBP Updating action hierarchy
Updating action hierarchy
        for all actions in model set
                processing action lay pile1
                        lay pile1 is an abstract action
                        lay pile1. object = pile1
                        pile1. super component = the foundations
                        the foundations. abstract action = lay
                        making lay pile1 a sub action of lay foundations
                processing action set out position pile
                        set out position pile1 is a primitive action
                        pile1. abstract action = lay pile1
                        making set out position pile1 a sub action of lay pile1
                .....

--- MBP actin hierarchy updated, instance trace

instance lay1
        instance of class lay
        object    the foundations
        main effects the foundations laid = true
        sub       {build supermarket}
        Sub {lay pile1, lay beam1, lay beam2}

instance lay2
        instance of class lay
        object    pile1
        main effects pile1 = laid
        sub       {lay foundations}
        Sub {set out position, lay pile mat etc.}

instance lay pile mat1
        instance of class lay pile mat
        object    pile 1
        main effects pile mat = laid
        sub{lay pile1}
        Sub {null}
```

With activity synthesis complete and actions ordered into a hierarchy, activities may be synthesised.

```
--- MBP generate activities
  for all actions
        action lay1
                generate instance of class activity :activity1
                activity1. action = lay (lay1.action)
                activity1. object = pile 1(lay1.object)
        .....
-- MBP activity trace

acity1
        object    pile1
        action    lay
```

With activity generation complete, the MBP invokes the generate dependency algorithm for each instance of class concept within the domain model.

```
--- MBP generating dependency
  for all objects
        processing object supermarket
              generate dependency returns null
        processing object the foundations
              generate dependency returns null
        processing object the roof
              generate dependency returns null
        processing object beam1
              link abstract action with main effects (supported by)
              beam1 abstract action = lay 1
              lay1. activity = activity6
              Supported by set = {pile1, pile2}
              supported by abstract action set = {lay2, lay3}
              supported by activity set = {activity4, activy5}
              link {activty4, activty5} before activity 6
              primitive dependency (set out position, prepare ground)
              beam1.set out position. activity = activity 56
              beam1.prepare ground activity = activity 44
              link activity 56 before activity 44
              .....
              ....

-- MBP trace of activities with dependency assigned.

activity1
        object    pile1
        actin     lay
        dependant upon = null
        dependant upon me = activity2,
activity2
        object    beam1
        actin     lay
        dependant upon = activity1
        dependent upon me = activity3     --- object lay roof steelwork
```

The second phase of dependency synthesis considers the *aggregate conditions* within the model. Only the instance of class *ACTION lay foundations* contains an aggregate condition: *aggregate condition (safe-site)*. A trace of the MBP processing this action is depicted below.

```
--- MBP processing aggregate conditions
  processing preconditions lay the foundations
        aggregate condition = safe site.
        activating inference package safe site
        returns true and list {site fencing erected, school safety lecture given}
        appending  condition  list to precondition value facet of lay the foundations

--- MBP trace of actions effected by aggregate conditions
        action lay the foundations
        precondition: value site fencing = erected, school safety lecture = given
        if_needed  aggregate condition (safe site)
--- MBP v4 complete 13:09:23
```

With aggregate dependency determined, the MBP process is complete.

7.5 HTN planner interface

The MBP generates actions and dependency constraints based upon the properties of components and the relationships between components. HTN planning combines task networks containing action and dependency knowledge to form a complete and consistent plan with condition and effect constraints established and maintained. This section describes how the results of the MBP process may be complied into task networks and passed to a HTN planner. The description first identifies the point at which the two technologies may be integrated, before detailing the task network compilation algorithms. The description concludes with a trace of the HTN planner interface processing the results of the MBP process generated in section 7.4.

7.5.1 Interface point

A planning problem is specified to a HTN planner through a non-primitive task. This task is then refined until a plan is synthesised containing only primitive actions and no action interactions (see Chapter 2 and Chapter 3 for a detailed description of HTN planning). HTN planners identify refinements for tasks through requests to a schema library. These requests are of the form *expand {pattern}*. Where *pattern* is of the form *function arg1..argn*. For example, *expand {build house}*. At this point the interface may compile task networks from the results of the MBP process into a format accepted by a HTN planner.

The integrated architecture places a control flag within the schema library which the planning system user may set to determine if the library is run in traditional mode, model-based mode, or in a combination of both modes. This approach permits the HTN planner component of the integrated architecture to be executed with schema selection performed unmodified, from the results of the MBP, or from a combination of both sources. Figure 7-25, below, depicts the interface mode selection algorithm.

```
mode flag : Static | Dynamic | Mixed

expand {pattern}
    if mode flag = Dynamic
        return list of schemas from model
    else if flag set to Static
        return list of schemas from library
    else If flag set to Mixed
        return list of schemas from model and schema library
    end if
```

Figure 7-25, Interface mode selection algorithm

The rationale for providing different interface execution modes is described in Chapter 8 and critiqued in Chapter 9

7.5.2 Task-Definition

The definition of the problem a HTN planner is to achieve is traditionally supplied through a task definition file. The problem definition of the results of the MBP processing may be defined as the concatenation of the abstract action of the instance a model's project *CLASS* and the name of that project *CLASS* instance. For example, in description synthesised in section 7.4.3, the task definition would be *build the-supermarket*. A *Task-Definition* algorithm is provided to automatically synthesise this definition. The task definition template instantiated by the *Task-Definition* algorithm is depicted in Figure 7-26 below.

```
task "project. abstract action" # "project";
  nodes    1 start,
           2 finish,
           3 action {project.abstract-action  project};
  orderings 1-->3,3-->2;
end schema;
```

Figure 7-26, Task definition template

7.5.3 Integrated task expand algorithm

The integrated architecture's task expand algorithm intercepts requests from the HTN planner component of the integrated architecture to provide candidate schemas for expanding a task. The algorithm compiles the hierarchical task network to satisfy requests from the knowledge structures synthesised from the MBP component. The algorithm is defined in Figure 7-27 below.

The *HTN-expand* algorithm accepts two parameters from a HTN expand request: *function* and *argument*. The algorithm constrains *function* to be an instance of class *ACTION* and *argument* to be an instance of either class *PROJECT,* or class *COMPOSITE-OBJECT*, or class *PRIMITIVE-OBJECT*. The constraints ensure the semantics of the expand request *action object* are maintained.

The *function* parameter is further constrained to be related to the *argument* parameter through an *abstract-action* relation. This constraint first ensures that the action is to be applied to the object, and second that the algorithm does not attempt to expand a *primitive-action.*

If the parameters are consistent, the algorithm first creates a new schema with a name generated by concatenating the *function* and *argument* parameters. The *expands* field of the new schema is also set equal to this concatenation, thus, defining the task for which the new schema is an suitable refinement.

```
HTN-expand (function, argument)

  ;;; parameter constraints
  argument    =   Instance of class PROJECT or instance of class COMPOSITE-OBJECT or
                  instance of class PRIMITIVE-OBJECT
  function    =   instance of class ACTION related to argument through the abstract-action
                  relation.
  ;;; algorithm
  new-schema-name = function #' '# argument
  new-schema-name.expands = new-schema-name

  ;;; synthesis nodes
  if argument is an instance of class PROJECT or COMPOSITE-OBJECT then
      let object-list = subcomponents of function
      new-schema-name.nodes = the abstract actions of object-list # the names of
      the objects within object-list
  else if argument is an instance of class PRIMITIVE-OBJECT then
      new-schema-name.nodes = the primitive actions associated with argument # argument
  end if

  ;;; process main effects and side-effects
  for each action added to new-schema-name.nodes
      add main-effects to new-schema.only_use_for_effects including "at ?node-number"
      where ?node-number = position of current new-schema-name.nodes in list add side-
      effects to new-schema.effects including "at ?node-number"
      where ?node-number = position of current new-schema-name.nodes in list
  end loop

  ;;;process conditions
  for each action added to new-schema-name.nodes
      copy all actions preconditions into new-schema-name.conditions
      if precondition is as a result of an ordering constraint between actions
          make the condition supervised if producing action is within new-
          schema-name.nodes and add an ordering constraint to the schema
          otherwise make condition an encapsulation-busting condition
      end if
  end loop
end HTN-expand.
```

Figure 7-27, HTN-expand algorithm

The generation of nodes may take one of two paths. First, if the *argument* parameter corresponds to a instance of class *PROJECT* or class *COMPOSITE-OBJECT* then the nodes of the new schema become equal to the abstract action of each sub component of the *argument* parameter combined with the name of each sub component. Second, if the *argument* parameter corresponds to an instance of class *PRIMITIVE-OBJECT* then the nodes of the new schema become equal to the primitive actions of the *argument* parameter combined with the name of the *argument* parameter.

Effects are instantiated in the new schema by copying over the *main-effects* and *side-effects* of the actions included within the new schema into the *only_use_for_effects* and *effects* fields respectively.

Condition compilation is divided into a number of stages. First, the preconditions of each action included within the node field of the new schema are copied into the schema's conditions field. Each precondition is then processed to determine its type. If a precondition results from an ordering relationship between activities in data structures generated by the MBP, the producing and consuming action of the condition are known. However, the producing action may not be a member of the current schema, hence, a *supervised* condition may not be placed. If the producing action is a member of the current schema a supervised condition and an ordering constraint is added to the new schema to complete processing of the condition. If, however, the producing action is not a member of the current schema, the condition is typed as an *encapsulation-buster*. This condition type indicates that the producing and consuming action are known, but the constraint cannot be added to a HTN plan until both the producing and consuming actions have been inserted into the plan. The processing of *encapsulation-buster* conditions is detailed in section 7.5.4 below.

Figure 7-28 below presents a trace of the HTN-expand algorithm executing over the domain description produced by the MBP in section 7.4.3.

```
HTN-planner interface invoked
  generating task
      project = supermarket
      project.abstract-action = build
      generating task definition schema
          :task build-supermarket-task-definition
          :  nodes     1 start,
          :    2 end,
          :    3 action (build supermarket);
          :  orderings 1-->3,3-->2;
          :end task;;; build-supermarket

HTN Planner invoked with task build-supermarket.
  issues {action {build supermarket}}
  picked issue {action (build supermarket}
  request HTN-expand (build supermarket)
  new-schema-name = build-supermarket
  supermarket is an instance of class PROJECT
  new-schema-name.nodes =          [1] action {lay foundations}
                                   [2] action {erect roof}
  new-schema.only_use_for_effects =    foundations laid at [1]
                                       roof erected at [2]
  no orderings
  no conditions
  return schema
      :schema build-supermarket
      expands (build supermarket)
      nodes     1 .action (lay foundations),   ;;; from Sub of supermarket
          2 .action(erect roof);                   ;;; from Sub of supermarket
      effects   foundations  = laid at     [1]
                roof         =erected at   [2]
      end schema;;; build-supermarket
  HTN planner applying schema build-supermarket
  issues = {action{lay foundations}, action {erect roof}
  picked issue {action {lay foundations}}
  ....
```

Figure 7-28, Trace of the HTN-expand algorithm produced within the construction domain

7.5.4 Processing encapsulation-buster conditions

The *encapsulation-buster* condition type is included within the integrated architecture to relax the encapsulation constraint upon schemas. Within traditional HTN planning, a condition of type *supervised* may only be placed between actions which appear within the same schema. By following this constraint, HTN planners ensure that a schema does not make assumptions about the other actions which will be included within a plan. The *unsupervised* condition type permits a condition to be specified where its establishment may come from an action inserted by another schema, however, the condition type does not specify the specific action from which establishment will be achieved. Identifying the action from which a *unsupervised* condition will be established is left to the HTN planning engine.

In the context of the integrated architecture, the MBP component may identify ordering relations between actions which when compiled into task networks will not reside in the same schema. If the integrated architecture is executed in *integrated mode* only, then it is possible to guarantee that both actions will be included within a plan. Using an *unsupervised* condition type would therefore place an unnecessary overhead upon the HTN component of the architecture. The *encapsulation-buster* condition type is included to address this case.

The *HTN-expand* algorithm specified in Figure 7-27 above details the criteria under which an *encapsulation-buster* condition is placed. Conditions of this type are of the form {*producing action name, consuming action name, conditions*}. For example, {*lay-pile-action7, erect-steelwork-action3, pile7 = laid*}. When a schema is received by a HTN planner containing *encapsulation-buster* conditions, each *encapsulation-buster* condition is added to the *encapsulation-buster-queue*. When expansion of the schema is completed, the *consuming action* is guaranteed to be present within the plan as the schema containing the *encapsulation-buster* condition will always contain this action. Immediately after expanding the schema, the task refinement engine is modified to examine each member of the *encapsulation-buster-queue*.

If the *producing action* is present within the plan, the condition may be implemented by adding an ordering constraint between the *producing action* and *consuming action* and a *supervised* condition between the two actions corresponding to the *conditions* field within the condition. The *encapsulation-buster* condition is then removed from the queue.

If the *producing action* is not present within the plan, the condition cannot be processed. The condition is left on the queue, and the consuming action marked as *suspended*. This *suspension* prevents further refinement of the consuming action before the *encapsulation-buster* condition upon it has been recorded.

7.6 Implementation status

The integrated architecture described within the chapter has been implemented in Intellicorps KAPPA-PC. The implementation utilises the HTN workbench described in Chapter 3.

7.7 Summary and conclusions

This chapter presented an integrated architecture developed to exploit the relative capabilities of classical and model-based planning technologies. The architecture is composed of five components: a set of model-based domain modelling constructs, a domain model - model-based planner interface, a model-based planner, a model-based planner - HTN planner interface, and a HTN planner.

The model-based domain modelling constructs provide the mechanism for modelling a specific application domain in terms of object, actions, and interrelationships between objects. The model - model-based planner interface defines the constraints upon the modelling constructs expected by the model-based planner component. The model-based planner processes an application domain representation to synthesise the actions, conditions, effects, and dependency constraints required by a plan. The model-based planner - HTN planner interface compiles the results of the model-based planning process into a format which may be input to a HTN planner. The HTN planner assembles the results of the model-based planning process into a complete plan, detecting and resolving action interactions.

In conclusion, this chapter has presented an architecture which may be implemented to synthesise plans utilising model-based and classical planning techniques. However, the ability of this architecture to address the complementary strengths and limitations between the technologies identified in Chapter 4 and Chapter 6 has not been assessed. This chapter is complemented by Chapter 8 which seeks to answer this question in addition to assessing the architecture's commercial applicability.

Blank

8. Evaluating the integrated architecture

There is nothing either good or bad, but thinking makes it so.

William Shakespeare (1564–1616)

8.1 Introduction

Chapter 7 defined a new planning architecture designed to exploit the capabilities of classical HTN and model-based technologies. This chapter evaluates this integrated architecture against the rationale for its development identified in Chapter 4 and Chapter 6, and assess its commercial utility. Specifically, this chapter evaluates the architecture from each of the following perspectives:

- **Strengths and limitations.** The limitations identified with HTN and model-based planning in Chapter 4 and Chapter 6 are examined, and the facet(s) of the integrated architecture which address each issue identified.

- **Literature examples**. A representation of the Pacifica military evacuation domain (summarised within Appendix C) is derived to evaluate the generality of the integrated architecture.

- **Automated planning expert.** Reviews from automated planning experts obtained through conference submissions and publications are summarised and answered.

- **Industrial planning expert.** The comments received from people involved in the generation and use of plans within the Llewellyn Group of Companies are summarised and answered to provide an industrial perspective on the integrated architecture.

Each evaluation perspective is applied in turn below. The chapter concludes by summarising the utility of the integrated architecture in the context of the perspectives above.

8.2 Strengths and limitations perspective

Chapter 4 and Chapter 6 identified a number of complementary strengths between classical and model-based planning technologies. The strengths and limitations evaluation perspective considers each limitation and identifies the facet(s) of the integrated architecture which addresses it.

8.2.1 Expressiveness - HTN planning

8.2.1.1 Variable nodes

The need to vary the number of nodes within task networks to adapt to specific problems was identified within the construction and military evacuation domains analysed in Chapters 4 and 6. Within the integrated architecture, the function of generating schemas with an appropriate number of nodes for a given problem is achieved through the *aggregation* and *abstract action* constructs. The instance diagram below (Figure 8-1) represents a component with a variable number of subcomponents. Each subcomponent has one abstract action associated. The task network compilation process responds to requests of the form *expand {component.abstract-action component}* by returning the concatenation of each of the components subcomponents' name and abstract action. The task network resulting from such a request on Figure 8-1 is depicted in Figure 8-2.

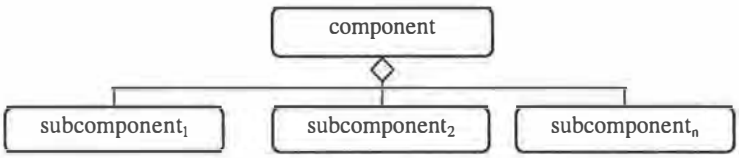


Figure 8-1, Aggregation relation

```
schema component.abstract-action # component;
    expands {component.abstract-action component};
    nodes  1.  action { subcomponent1.abstract-action subcomponent11,
           2.  action { sub-compoenent2.abstract-action subcomponent21,
           ...
           n.  action { sub-compoenentn.absract-action subcomponentn1};
end schema;
```

Figure 8-2, Schema compiled from the objects in Figure 8-1

The O-Plan task refinement planner provides two constructs to support the type of functionality described above: *foreach*, and *iterate*. The syntax of the *foreach* construct is depicted in Figure 8-3 below.

```
N foreach action(action-name ?parameter)
        for ?parameter over {set}
```

Figure 8-3, Task formalism *foreach* construct

The *foreach* construct generates a task network containing a node *action-name ?parameter* for each member of the *set* specified in the *for over* line. The term *?parameter* is instantiated at each node to the element of *set* it is included for. For example, if the *set* is defined as {$member_1$, $member_2$, $member_3$), the construct will generate a schema with the three nodes: *action-name* $member_1$, *action-name* $member_2$, and *action-name* $member_3$. The *iterate* construct differs from *foreach* only in the ordering constraints placed between the nodes. Action are left unordered with respect to each other in the *foreach* case, and are ordered sequentially as they are generated in the *iterate* case.

Figure 8-4, below, encodes an approximation of the integrated architectures variable node function using O-Plan's *foreach* construct.

```
1. schema integrated-architecture-simulate;
2.   expands (generic-action ?object);
3.          N for each action (generic-action ?new-object)
4.                for ?new-object over {?object sub};
5. end schema;
```

Figure 8-4, Task formalism approximation of the integrated architecture's variable nodes function

The schema *integrated-architecture-simulate* refines tasks of the form *generic-action ?object* by generating a node for each subcomponent of the parameter *?object*. To demonstrate the schema's function, consider the fragment of the construction domain specified in Figure 8-5, below. The fragment is specified in a planner's always context (i.e., facts which do not change during planning). The term *the-supermarket* is an instance of the class *PROJECT* with a single attribute *sub*. The *sub* attribute lists the instances which are subcomponents of *the-supermarket* (assume that each member of the set *sub* is defined as being of class *COMPOSITE-OBJECT* or *PRIMITIVE-OBJECT*).

```
initially    the-supermarket : PROJECT
             sub the-supermarket {the-foundations, the-roof, the-windows-and-doors}
```

Figure 8-5, Example of sub retaliation in the construction domain

If a HTN planner is given the initial task of *generic-action the-supermarket*, the schema *integrated-architecture-simulate* will match with this task through the *expands* construct on line 2. The *foreach* statement on line 3 will generate a new *generic-action ?new-object* node for each subcomponent of *the-supermarket*. The resultant schema is depicted in Figure 8-6 below.

```
schema integrated-architecture-simulate;
        expands (generic-action the-supermarket);
        nodes  1 action {generic-action the-foundations},
               2 action {generic-action the-roof},
               3 action {generic-action the-windows-and-doors};
end-schema;
```

Figure 8-6, Result of applying schema *integrated-architecture-simulate* to the construction problem

The *integrated-architecture-simulate* encoding contains a number of limiting issues. First, the action name *generic-action*, as opposed to ideal action name (i.e. *lay* in the case of *the-foundations, erect* in the case of *the-roof,* and *install* in the case of *the-windows- and-doors*), results from the restriction upon the *foreach* construct that the construct may work over one action name only. The *foreach* construct cannot vary the action names it generates by taking into account the type of the object which it is considering. If the construct was modified to take this factor into account, the resultant actions would not match the *integrated-architecture-simulate* expansion criteria of *generic-action ?object*. Hence, the new actions generated by *integrated-architecture-simulate* would not be applicable for further refinement by the schema.

The second limiting issue with *integrated-architecture-simulate* is the termination criteria of the construct. This issue and the first *generic-action* problem may be demonstrated through a generic example. Consider the type and component specifications in Figure 8-7 below. Five domain objects are defined (x, y, z, a, b). The domain objects are related though the subcomponent relation so that y and z are subcomponents of x, and a and b are subcomponents of y. Component z has no subcomponents. Each component is of a different type ($TYPE_1$, ..., $TYPE_5$). The action name which would ideally prefix each component depends upon the component's type. This ideal name will be referred to as $TYPE_N$.*ideal-action-name*. In a specific example, objects of type *FOUNDATIONS* should be prefixed with the action name *lay*, whilst objects of type *WINDOW-AND-DOORS* should be prefixed with the action name *install*.

```
x: TYPE1, y: TYPE2, z: TYPE3, a: TYPE4, b: TYPE5
x.sub = {y, z}
y.sub = {a, b}
```

Figure 8-7, Generic problem specification

For the problem specified in Figure 8-7, a HTN planner would be initialised with the following initial task.

```
initial task definition = action {generic-name x}
```

Component *x* is the project level component (i.e. it is not the subcomponent of any other component), and *generic-name x* will match against the *integrated-architecture-simulate* schema's expands criteria. The initial task becomes the only item on the HTN planner's agenda. The action name *generic-name* as opposed to $TYPE_1$*.ideal-action-name* must be used so that the HTN expansion routine will match the task's name with the *integrated-architecture-simulate* schema. The processing of this first task is depicted in Figure 8-8, below.

```
agenda = (action {generic-name x})
pick agenda item action{generic-name x}
request schema which matches selected agenda item.
result is integrated-architecture-simulate with parameter ?parameter instantiated to x
        N for each action (generic-action ?parameter)
            for ?parameter over {x.sub}
applying integrated-architecture-simulate results in the following schema

    schema integrated-architecture-simulate
        nodes       1. action {generic-name y},
                    2. action {generic-name z};
    end schema;
```

Figure 8-8, Applying integrated-architecture-simulate to component *x*

Applying *integrated-architecture-simulate* to concept *x* results in two new agenda items. The processing of one of the new items is depicted below:

```
agenda = (action {generic-name y}, action {generic-name z})
pick flaw action{generic-name y}
request schema which matches selected flaw.
result is integrated-architecture-simulate with parameter ?parameter instantiated to y
        N for each action (generic-action ?parameter)
            for ?parameter over {y.sub}
applying integrated-architecture-simulate results in the following nodes
        1. action {generic-name a}
        2. action {generic-name b}
```

Figure 8-9, Applying integrated-architecture-simulate to component *y*

This process will continue until the planner reaches components which do not have subcomponents. Figure 8-10, below, depicts the case of component *z* which has no subcomponents.

```
agenda =   (action {generic-name z}, action {generic-name a},
           action {generic-name b)
pick flaw action{generic-name z}
request schema which matches selected flaw.
result is integrated-architecture-simulate with parameter ?parameter instantiated to y
        N for each action (generic-action ?parameter)
            for ?parameter over {z.sub}
            applying integrated-architecture-simulate results in the following nodes
        NULL
```

Figure 8-10, Applying integrated-architecture-simulate to component z

As component *z* has no subcomponents, the *integrated-architecture-simulate* schema produces no new nodes for component *z*. The result is a schema will the nodes set to null. Hence, it is not possible to write an encoding in an existing HTN formalism which simulates the model-based planning algorithm. The need to include actions based upon the type of a set member not possible. It is not possible to write a generic schema which terminates when a component with no subcomponents is reached.

A trace of the model-based planner solving the same problem is depicted below. Note that the action names generated take into account the type of component over which they are working and that it terminates on the primitive level components, i.e. components with no subcomponents.

```
task = x
generate x.abstract-action-name
generate x.abstract-action-name.effects
for each sub of x and then for each sub of sub - terminating when hit a primitive
    generate y.sub.abstract-action-name
    generate y.sub.abstract-action-name.effects
    generate z.sub.abstract-action-name
    generate z.sub.abstract-action-name.effects
    generate z.sub.primitive-action-name
    generate z.sub.primitive-action-name.effects
    generate a.sub.abstract-action-name
    generate a.sub.abstract-action-name.effects
    generate a.sub.primitive-action-name
    generate a.sub.primitive-action-name.effects
    generate b.sub.abstract-action-name
    generate b.sub.abstract-action-name.effects
    generate b.sub.primitive-action-name
    generate b.sub.primitive-action-name.effects

end
```

In conclusion, the integrated architecture exploits the subcomponent relation within a domain to generate the actions which account for both the number of components in a specific problem instance and the type of those components. An equivalent functionality cannot be produced in current implementations of HTN planning. As described above, HTN planners cannot consider the action name which should be associated with a domain concept, nor can action synthesis terminate when components without subcomponents are reached.

8.2.1.2 Conditional nodes

The variable node mechanism generates abstract actions accounting for the variable number of components and their types within a problem instance. The conditional nodes mechanism generates the actions required by instances of class *PRIMITIVE-OBJECT* within a problem instance. This distinction between variable and condition node reasoning may be illustrated through the domain model depicted in Figure 8-11, below.

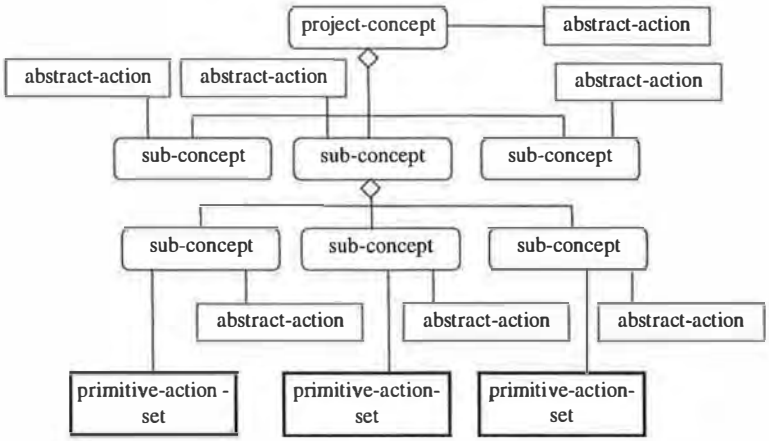


Figure 8-11, Position of conditional node reasoning with the domain model

The variable node reasoning navigates through the model generating the abstract actions and, when a primitive object is reached, activating the conditional node reasoning for the primitive action sets within a model. The conditional node reasoning, activated by the variable node reasoning process, determines which members of the set of possible primitive actions associated with a component will be instantiated.

Conditional node reasoning was motivated by two observations. First, that it is not practical to encode all possible methods for achieving a task. Second, that the syntax of filter conditions is difficult to map to the knowledge within a domain for determining an action's inclusion. Both these limitations are justifiable within the context of a formalism designed to manage the computational complexity encountered whilst working over a partially-ordered partially instantiated plan. The integrated architecture, however, exploits the planning which may be achieved within the space of a static domain model. A more expressive formalism may therefore be used.

The existing HTN encoding approach results in a number of actions sets or methods, with each set's applicability specified by a number of filter conditions. The integrated architecture specifies a single set of actions for achieving each object within a domain. Each action may be associated with knowledge for determining if it should be associated with a specific object. The possible set of actions for achieving the *BEAM* class is depicted in Figure 8-12 below.

```
class BEAM
primitive-actions: if-needed {
        must (a, SET-OUT-POSITION)
        must (a, EXCAVATE-BEAM)
        must (a, BLIND-BOTTOM)
        ...
        infer(a, MOULD-OIL, determine-mould-oil)
        infer(a, STRIKE-FORMWORK, determine-strike-formwork)
        }
```

Figure 8-12, class BEAM's primitive action specification

Set membership for a specific instance is determined through the instantiation directives *must* and *infer* which prefix each action in the possible set. *Infer* invokes a inference package to determine an actions inclusion. *Must* is a special case of the *infer* directive which always determines that an action should be associated. Figure 8-13 summarises the mechanisms which the *infer* directive may utilise. The *infer* directive is implemented through a instantiation directive handler. The handler parses the directive's parameters (the text in brackets following the directive name), identifies the action class to which it implies (*STRIKE-FORMWORK* in the figure below), and the inference package which should be invoked to determine the actions relevance (*infer-strike-formwork* in the figure below).

Infer (a, STRIKE-FORMWORK, infer-strike-formwork)

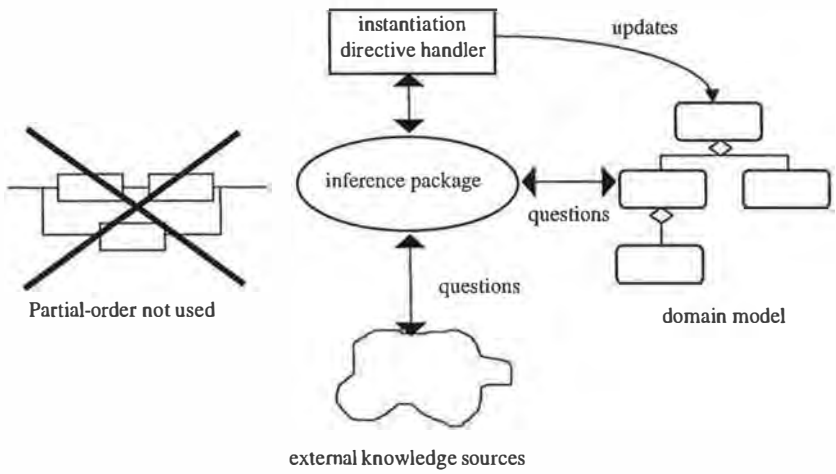


Figure 8-13, Inference directive

Instantiation directives process the results of an inference package. In the case of *infer,* the directive will instantiate an instance of the action class specified in the parameter if and only if the inference package returns true. If the inference package returns false, no action is taken.

Figure 8-13 depicts the constraints upon an inference package's reasoning. The inference package may consult knowledge about a domain encoded in the domain model (attribute, instances etc.), and may connect to external data-bases and other knowledge sources. The mechanism is constrained not to derive any information from a partially instantiated partial-order plan.

An equivalent HTN encoding may be achieved using the following framework. The possible set of actions which may be associated with a concept may be written as a single schema. Actions which would be prefixed with the *must* directive are written as *primitive actions* (i.e. actions with no further refinement). Actions which would be prefixed with the *infer* directive are written as *non-primitive* actions (i.e. action which require further refinement). The HTN representation of the model-based *BEAM* action set is depicted in Figure 8-14 below.

```
schema emulate-BEAM-primitive-actions;
    vars        ?beam : BEAM;
    expands     {lay ?beam};
    nodes  1    primitive       {set-out-position ?beam},
           2    primitive       {excavate ?beam},
           3    primitive       {blind-bottom ?beam},
           4    action          {mould-oil ?beam},
           5    action          {strike-formwork ?beam};
end schema;
```

Figure 8-14, HTN conditional nodes functionality - stage 1

Each of the non-primitive actions is provided with two refinements. The first refinement represents the case of when the action should be included, and the second when the action should not be included. Figure 8-15 presents the two methods for refining the *mould-oil ?beam* task within Figure 8-14.

```
schema mould-oil-yes;
        vars        ?beam : BEAM;
        expands     {mould-oil ?beam};
        nodes       1  primitive {pour-mould-oil ?beam};
        only_use_if ?beam formwork = custom;
end schema;

Schema mould-oil-no;
        vars        ?beam: BEAM;
        expands     {mould-oil ?beam};
        nodes       1  dummy;
        only_use_if ?beam formwork = prefabricated;
end schema;
```

Figure 8-15, HTN conditional nodes functionality - stage 2

The encoding of knowledge for determining if an action should be included within a plan is achieved in HTN planning through filter conditions. The syntax of filter conditions was demonstrated within Chapter 4 to be difficult to map to the domain

knowledge found within industrial problems. Planners such as UCPOP and O-Plan, however, provide mechanisms for linking to external knowledge sources in a way similar to the inference package mechanism supported by the integrated architecture. UCPOP provides a mechanism known as *facts* and the O-Plan system a *compute conditions* mechanism. Each mechanism is defined below, before a comparison with the integrated architecture is provided.

The generic structure of a UCPOP *fact* (Barrett et. al. 1994, pp 11) is depicted below. The * notation implies repetition i.e., the <*variable-name*> parameter may be repeated one or more times.

```
(define (fact (<predicate-name> <variable-name>*))
        <function body>)
```

Facts permit the evaluation of predicates to be implemented via a user defined procedure; in the current UCPOP implementation, these procedures must be implemented within the Lisp language. Considering an example fact, the predicate *less-than* may be defined with two variables *?x* and *?y*. The function body may then be implemented with the Lisp function *(< ?x ?y)*. Hence, predicates of the form *less-than(3 4)* will be evaluated using the Lisp language function as opposed to the Modal Truth Criteria's backward search through a plan. Facts may return *predicate undefined, insufficient information to evaluate predicate* or *predicate defined* The first response indicates that it is not possible to evaluate the predicate, the second response that the predicate should be considered latter when more information is available. The third response indicates that the predicate has been successfully evaluated. A number of variable binding constraints may also be returned, and the planner permitted to nondeterministically chose constraints to add to a plan variable in order to establish the fact.

The O-Plan system provides *compute conditions* (Tate, Drabble, & Dalton 1994b, pp40) for linking domain descriptions to external sources of inference. An example compute condition from the *Pacifica* (Reece et. al. 1993) domain is depicted below (A full description of the Pacifica domain is provided in Appendix C).

```
(define transport (capacity e_left e_safe) ; --> c_safe c_left
        (let ((take (min e_safe capacity)))
        (list ( + e_left take)
            ( - e_safe take))))
```

The example compute condition above will take statements of the form *transport (?capacity, ?evacuees left, ?evacuees safe)*. An example instantiation would be *transport (50, 100,0)*. The Lisp function uses the capacity argument, in the example 50, to determine how many people may be evacuated. The number of people evacuated and the number remaining is then returned. In the example above, *(50, 50)* will be returned as 50 people may be evacuated, leaving 50 people still to be evacuated.

To date, the use of *facts* and *compute conditions* within demonstration domains definitions centres upon mathematical functions. These constructs, however, permit inference routines to be written for determining more complex domain facts. In the context of the beam laying example in Figure 8-15, a compute condition {*mould oil ?beam*} which returns *true* or *false* may be written. The condition could invoke rule-based reasoning to determine if a *mould oil* action was required for a specific beam. An *only_use_if* filter conditions may then be written, selecting between a *mould oil* action or a *null* action depending upon the compute conditions result. Reasoning of this complexity was implemented in Nonlin (Tate 1977) by making a three dimensional reasoning system available to the planner through the *compute condition* construct.

In conclusion, the *fact* and *computer condition* mechanisms supported by existing HTN planners may be used to implement the criteria for determining action inclusion via external knowledge sources. These constructs are comparable to the *infer* instantiation directive within the integrated architecture. However, task refinement planners do not provide the complementary mechanisms supported by the integrated architecture for determining the conditions, effects, and ordering constraints which result from the specific actions instantiated. It is the conditional nodes mechanism in combination with constructs for performing this assessment which leads to the integrated architecture being an advance on HTN only planning systems.

8.2.1.3 Conditional effects

Within the integrated architecture, conditional effects are achieved through two mechanisms. Implicitly, if an action is instantiated its effects will be considered for inclusion within a plan. Explicitly, if an action is instantiated, its effects are attached to actions using a similar mechanism to the attachment of actions to objects. Figure 8-16, below, depicts a fragment of a domain model with action assessment completed.

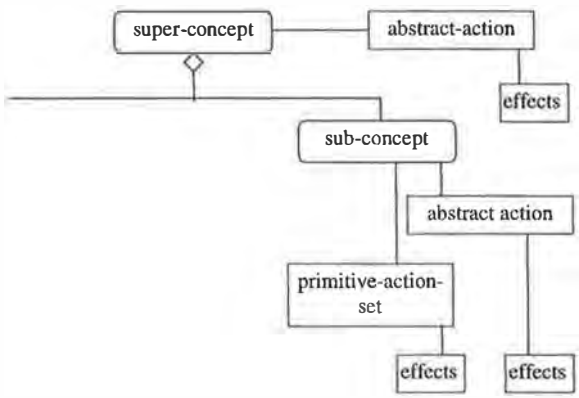


Figure 8-16, Position of effects within the domain model

The model-based planning algorithm recurses through the model instantiating abstract actions automatically, and primitive actions depending upon the results of inference packages. Each action (either abstract or primitive) has its *main-effects* and *side-effects* attributes assessed immediately upon instantiation. Effects are attached to attribute slots through two directives: *action-must,* and *action-infer*. In the latter case, inference packages will be invoked to determine if an effect should be attached to a specific action instance. In the earlier case, the effect is automatically attached.

Taking the first case of effect inclusion via the association with action, consider the example within Figure 8-17, below. The first schema, *schema-a,* would be produced by the model-based planner from a building's design containing one room with the attribute *room1.decoration* set equal to *tile*. Hence, the schema contains one effect, *room1 tiled*. The second schema, *schema-b,* would be produced where, in addition to room one, a second room was specified with the attribute *room2.decoration* set equal to *tile*. The schema contains the additional effect *room2 tiled* as a result of this design change.

```
schema-a;
    nodes   1. action {tile-room 1};

    only_use_for_effects
        room1 = tiled;
end schema;
```

```
schema-b;
        nodes     1. action {tile-room1},
                  2. action {tile-room2};
        only_use_for_effects
        room1 = tiled,
                  room2 = tiled;
end schema;
```

Figure 8-17, Example of effect inclusion through association with actions

Consider the case of the action class *TILE* being extended to include the effect *action infer (a, hazard ?object = ?inference result, infer_ceiling_hazard)*. The *infer_ceiling_hazrd* inference package contains reasoning to determine if working on a specific ceiling will cause a hazard and the type of that hazard. Using this mechanism, effects of the type defined in *schema-c* below may be derived. The schema contains the effect *hazard-room1 = asbestos* as a result of the inference package finding asbestos within *room1's* construction.

```
schema-c;
    nodes   1. action {tile room1},
            2. action {tile room2};
    only_use_for_effects
            room1 tiled,
            room2 tiled;
    effects
            hazard room1 asbestos;
end schema;
```

HTN and precondition planning systems provided two mechanisms for achieving conditional effects: *operator specification,* and *domain rules* or *axioms.* Each mechanism is described below, before the relationship with the integrated architecture's mechanisms is introduced.

The precondition achievement planner UCPOP supports conditional effects in operator specifications. Figure 8-18 presents an example UCPOP action with conditional effects. The operator has the semantics move item *?z* from location *?x* to location *?y*. The italicised effects *on ?z ?y ... clear ?x* are not conditional, the effects will always be asserted as a result of the operator to which they being inserted into a plan. Specifically, the object to be moved *?z* will always be on the designated location *?y* and it will no longer be at the initial location *?x*. The initial location *?x* will therefore always become clear.

```
(define (operator move)
        :parameters (?x ?y ?z)
        ...
        :effect  (and (on ?z ?y) (not (on ?z ?x)) (clear ?x)
        when (≠?y table) (not (clear ?y)))
```

Figure 8-18, Example UCPOP action with conditional effects

The bolded effect is conditional. The first clause *when((≠?y table)* is the condition under which this effect becomes applicable. If this condition holds, the effect *(not (clear ?y)* will be asserted. The semantics in this example are that if *?y*, the destination location, is a table then it will remain clear because a table will (under the assumptions in this encoding) support an infinite number of blocks. If, however, *?y* is not the table, then *?y*, the destination location of the object *?x*, will no longer be clear.

The second mechanism for providing conditional effects is known as *domain rules* in the SIPE and O-Plan[1] systems and *axioms* in the UCPOP system. Within this discussion domain rules and axioms will be referred to as domain rules. Domain rules differ from operator specified conditional effects in that the latter are attached to specific operators. Domain rules are written independently of operators. When an action is included in a plan, the planning engine (both HTN and precondition achievement) examines the world state. Domain rules are written with trigger mechanisms which, when matched against the world state, cause the effect attached to the rule to be asserted. Consider the domain rule *is-above* in Figure 8-19 taken from the UCPOP system. The *context* clause specifies that if an object *?x* is on another object *?y*, then this axiom is applicable. If the context statement holds as a result of a new action's inclusion within a plan, the *implies* part of the axiom is asserted as an additional effect of that new action. In the example below, the *on ?x ?y* condition results in the new fact *above ?x ?y* fact being asserted.

```
axiom is-above
    :context (on ?x ?y)
    :implies (above ?x ?y)
```

Figure 8-19, Example of the UCPOP Axiom construct

Domain rules introduce a number of issues to planning. For example, ensuring deductive closure and resolving contradicting effects. The SIPE system contains mechanisms for handling these issues whilst maintaining heuristic adequacy. See (Wilkins 1988, pp 90) for a detailed discussion of the issues surrounding the implementation of domain rules and the methods available for resolving these issues.

[1] Domain rules are not currently implemented in the O-Plan system.

In conclusion, the integrated architecture provides complementary effect assessment mechanisms to those provided by classical planning systems. The need for both quantified and conditional effects is well established within classical literature. The integrated architecture exploits the effect assessment which can be achieved through static domain knowledge, whilst classical planning supports the reasoning required to assess effects over an evolving world model. The integrated architecture combines the conditional and variable node reasoning to provide quantified effects and a domain's structure to ensure effects are compiled into the appropriate task refinement schemas.

8.2.1.4 Conditional dependency constraints - relationships

Dependency constraints between actions are synthesised within the integrated architecture through *relationships* and *aggregate conditions.* This subsection evaluates *relationship* dependency. *Aggregate* dependency is covered in section 8.2.1.5.

Relationship dependency is assessed from the relationships between instances in a domain model. Figure 8-20, below, depicts an instance, $instance_a$, related to two other instances, $instance_b$ and $instance_c$, through the relationship $relationship_r$.

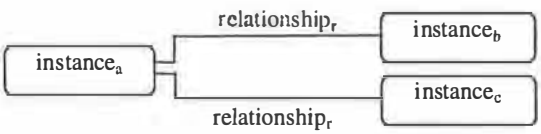


Figure 8-20, Example of domain specific relationships between model instances

Relationship dependency is assessed via the four constructs summarised below:

1. *For All I: instance related to self through relationship$_r$*, link self.abstract-action as dependent upon *I.abstract-action.* The *I.abstract action.main-effects* are copied into the precondition attribute of *self.abstract-action.*
2. *For All I: instance related to self through relationship$_r$*, link *self.abstract-action* as dependent upon *I.abstract-action.*
3. *For Some I:instance related to self through relationship$_r$* which match criteria *C*, link *self.abstract- action* as dependent upon *I.abstract-action.* Copy *I.abstract-action.main-effects* into the precondition attribute of *self.abstract-action.*
4. *For Some I:instance related to self through relationship$_r$* which match criteria *C*, link *self.abstract-action* as dependent upon *I.abstract-action.*

Figure 8-21, below, depicts a model of an abstract problem instance (*concept*$_a$, *concept*$_b$, *concept*$_v$, *concept*$_k$). The instances are structured through the *subcomponent* relation. The *abstract* and *primitive* actions of the instances *sub concept*$_v$ and *sub concept*$_k$ are depicted. *Concept*$_k$ is related to *concept*$_v$ through the relationship *relationship*$_r$. The results of applying each of the *relationship* constructs above to *concept*$_v$ in
Figure 8-21 are summarised below.

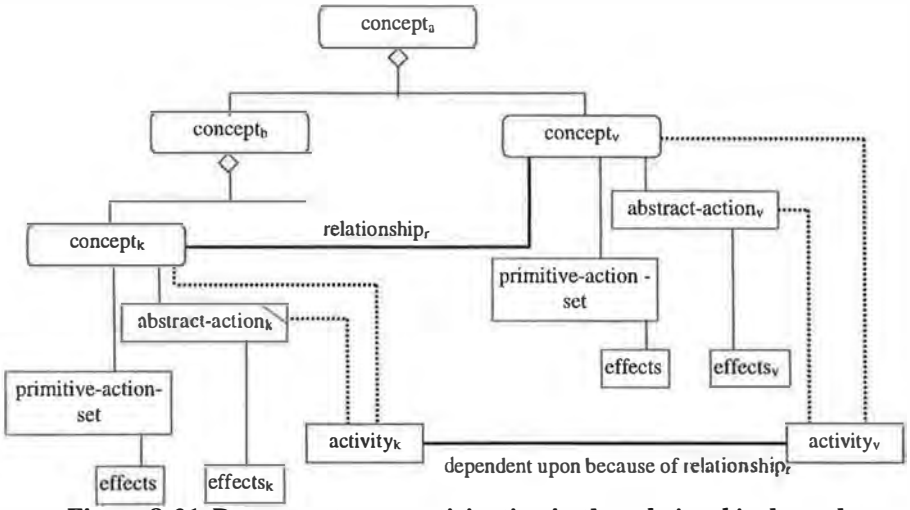


Figure 8-21, Data structures participating in the *relationship* dependency synthesis process

Applying relationship construct *1* to *concept*$_v$ with the parameter *relationship*$_r$ will first construct the set of concepts related to *concept*$_v$ through *relationship*$_r$. In the figure above, this set will include only *concept*$_k$. Construct *1* will link the activity associated with *concept*$_v$, *activity*$_v$, to be dependent upon the activity associated with *concept*$_k$. The *main-effects* of *abstract action*$_k$, *effects*$_k$, will be copied into the precondition attribute of *activity*$_v$.

Construct *2* will perform the same function as construct *1* with the exception of the copying of effects. Only a dependency relationship will be recorded between activities *activity*$_k$ and *activity*$_v$.

Construct *3* differs from construct *1* as inference packages will be used for determining the elements of the set of concepts related to *concept*$_v$ through relationship *relationship*$_r$ which will be set as predecessors. Inference packages are invoked to determine which elements will be made dependent. Once the subset is identified, the construct proceeds as construct *1*.

Construct *4* performs the same function as construct *3* with the exception of the copying of effects. Only ordering constraints are recorded between activities which pass the inference packages assessment criteria.

When *relationship* dependency is completed, the results are applied during the compilation of task networks. Ordering constraints and supervised conditions are added to resultant task networks as appropriate.
Traditional planning systems provide quantified preconditions and disjunctive preconditions. Each of these constructs is summarised below before being compared with the integrated architecture's relationship dependency functionality.

An example disjunctive precondition from the UCPOP system is depicted below. The precondition semantics are that this operator may be used if either *on ?x ?y* holds or *under ?x ?y* holds. If a planner wishes to make the operator applicable by achieving its preconditions, the planner may nondeterministically choose which of the preconditions to achieve.

```
:precondition (or (on ?x ?y) (under ?x ?y))
```

UCPOP permits quantification in preconditions. The example operator below defines the preconditions of an operating system's remove directory command.

```
:operator (delete ?directory)
:precondition forall (in ?file ?directory)
              (deleted ?file)
```

Both the MS-DOS and UNIX operating systems will remove a directory if and only if the directory contains no files. The *delete ?directory* operator contains a universally quantified precondition which generates a condition *deleted ?file* for each of the files recorded as *in ?file ?directory*.

Consider a new operating system command *really-delete*. The command deletes a directory and any files it contained, with the exception of system files. The UCPOP representation of this function is depicted below.

```
:operator (really-delete ?directory)
:precondition forall (in ?file ?directory)
              where (not system ?file)
                 (deleted ?file)
```

The *really-delete* operator creates an operator with a precondition set to all the non system files within a directory. The operator is providing existential quantification.

In conclusion, the integrated architecture provides a specialised form of quantified conditions which maps to the relationships between concepts in an application domain. The combination of conditional node, variable node, and conditional effect reasoning permits conditions to be placed between actions as a result of relationships between the components to which the actions related, and for this knowledge to be compiled into appropriate condition types within task refinement schemas. The *encapsulation-buster* condition type permits the causal structure of dependency constraints to be specified across the encapsulation unit of the schema, thus removing the need for a task refinement planner to establish a number of conditions.

8.2.1.5 Conditions - aggregate

The integrated architecture's aggregate condition function permits conditions to be specified in terms of the sub conditions which constitute them. Deriving the sub conditions is achieved through inference packages. Considering an example, the condition *site secure* in the construction industry domain will the sub conditions *3 meter fence erected* and *school safety lecture given* in one situation and the conditions *2 meter fence erected* in a second.

The integrated architecture makes the requirement of aggregate conditions explicit and provides an effective mechanism for their implementation.

8.2.1.6 Expressiveness argument conclusion

When considered in isolation, the new facets of the integrated architecture may either be achieved or almost achieved in existing classical planning technologies. It is, however, the ability of the new architecture to provide conditional nodes, variable nodes, quantified and conditional effects, and quantified and conditional conditions in a coherent form to synthesis task networks which distinguishes it form existing technologies.

8.2.2 Expressiveness - model-based planning

Traditional model-based planning systems do not incorporate the concept of conditions and effects. Without these concepts and a question and answering algorithm, it is not possible to either infer ordering constraints based upon the producer - consumer relationship between action effects and conditions nor detect and resolve interactions between these pairs.

The integrated architecture incorporates action conditions and effects within the MBP representation formalism and utilises traditional task refinement algorithms to both establish and maintain the causal structure of a plan.

8.2.3 Redundancy

Redundancy was identified in Chapters 4 and Chapter 6 between the encodings of multiple methods for performing a task, and similar methods for performing different tasks. The redundancy issue is addressed by the integrated architecture through two mechanisms.

First, the constructs described in the section 8.2.1 replace the method of statically encoding task networks. Each instance within a domain has its abstract actions, primitive actions, effects, conditions, and ordering constraints inferred. These constructs are then compiled to form task networks which meet specifically the needs of each concept instance. Thus, removing the need to specify multiple methods for performing a task. Methods are instead dynamically compiled to meet the requirements of a specific problem instance.

Second, the use of the inheritance relation within the object-oriented domain modelling scheme permits knowledge to be organised into a hierarchy. Consider the classes in Figure 8-22 below. Knowledge encoded within $CLASS_A$ is inherited by all the other classes in the figure. The knowledge from both $CLASS_A$ and $CLASS_B$ is inherited by $CLASS_D$. Hence, actions, attributes and dependency knowledge defined within $CLASS_A$ is utilised by all the other classes in the figure. Thus changes to $CLASS_A$ are reflected in the remainder of the system.

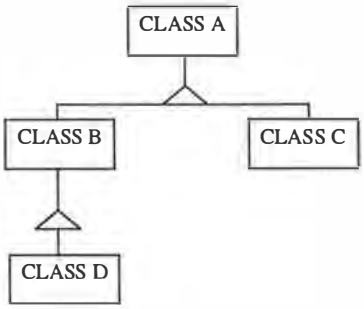


Figure 8-22, Class hierarchy

Inheritance permits a domain to be specified as a number of specialisations, with each class inheriting facets of its parent, and adding new facets unique to itself.

In conclusion, the domain rules and conditional effects within classical planning systems address in part the redundancy issue. The integrated architecture compliments these constructs by providing mechanisms to model a domain in terms of specialisations and for synthesising task networks to meet the requirements of a specific problem instance.

8.2.4 Semantic distance

Erol notes that the writing of domain specifications is "... the most neglected aspect of planning, and there is not an established software-engineering methodology to guide this job." (Erol 1995, pp91). Yet Chien notes that "... the amount of effort required to construct, debug, verify and maintain the planning knowledge base" (Chien 1996) is a major factor in deciding whether AI planning techniques are applicable to real-world applications.

Current work in this area is either providing tools for debugging and verifying planning knowledge bases (e.g. (Chien 1996)) or is providing conceptual links between planning formalisms and software - business engineering methodologies. An example of the latter work is the connection between the <I-N-OVA> model of plans as a set of constraints and the IDEF business process methodology (Tate 1996).

The KADS knowledge-based system development methodology and the OMT (Rumbaugh et. al. 1991), Booch (Booch 1991), and Syntropy (Cook & Daniels 1994)object-oriented methodologies are centred upon the concepts of objects, classes, inheritance, aggregation, and relationships. The methodologies provide a set of guidelines and techniques for identifying the constructs within a domain, notations for representing them, and tools for managing reviewing and validating results.

The integrated architecture is centred around the same constructs as KADS and object-oriented methodologies. The technique therefore provides constructs which map closely to existing elicitation and modelling methodologies and tools. This view is supported by McCluskey (McCluskey & Porteous 1995; 1996a; 1996b; McCluskey, Kitchin, & Porteous, 1996, Kitchen & McCluskey 1996). McCluskey compiles a precondition achievement action representation from the state chart notations supported by object-oriented methodologies under the rationale that such methodologies provide a tool supported representation which maps closely to domain experts' knowledge.

8.3 Literature examples perspective

The literature examples perspective aims to demonstrate the generality of the integrated architecture. The architecture's development and testing prior to this chapter has been centred within the construction domain.

Drabble and Tate (1994) define a continuum of applications ranging from resource intensive problems like job shop scheduling and condition and effect intensive problems such as blocks world. The authors position their O-Plan system midway on this continuum where, the authors claim, a significant number of real world applications rest.

The Pacifica (Reece et. al 1993) military evacuation domain is used within this section to demonstrate the generality of the integrated architect as the domain provides an example ranging from the middle to the condition and effective intensive points on Drabble and Tate's continuum.

8.3.1 Pacifica

The Pacifica domain (Reece et. al. 1993) is described in Appendix C. The domain centres upon the non-combatant military evacuation of a geographical area.
The encoding of the Pacifica domain in the model-based formalism is divided into two stages. First, section 8.3.1.1 defines the mapping of the overall structure of the domain. Second, sections 8.3.1.2 and 8.3.1.3 step through in detail the representation and planning of two central phases of an evacuation mission.

8.3.1.1 Mapping the structure of pacifica to the model-based formalism

Pacifica contains the following decision points:

- Number of air and ground transports available to a mission.
- Number of cargo craft available to a mission.
- Assignment of air and ground transports to cargo craft.
- Arrangement of loading, take off, landing, and unloading of cargo aircraft.
- Assignment of air and ground transports to outlying cities for effecting the evacuation
- Reloading of air and ground transport onto cargo craft for returning to safety after the mission.
- Returning evacuees to safety.

The encoding of Pacifica described in this chapter makes the following assumptions about the information supplied to the planning system by the system's user.

- The number of ground transports, air transports, and cargo aircraft available for a specific evacuation mission is specified.
- The assignment of air and ground transports to cargo aircraft is specified.
- The number of people to be evacuated from each city is defined.

An example problem definition is depicted in Figure 8-23 below.

```
;; ground, air, and cargo transports available for mission.
gt1, gt2, gt3: GROUND_TRANSPORT
at1, at2, at3: AIR_TRANSPORT
c130, c140 : CARGO_AIRCRAFT
cityA, cityB :CITY
;; assignment of cargo to cargo transports
c130.carries = at1, at2, at3
c140.carries = gt1, gt2, gt3

;; cities to be evacuated and the number of evacuees in each city
cityA.NumberOfEvacuees = 10
cityB.NumberOfEvacuees = 11
```

Figure 8-23, Initial Pacifica problem definition

Figure 8-24, below, presents the first level domain model of the domain. The *EVACUATION-OPERATION* class is related to three sub classes: *LOCATE-EQUIPMENT, EVACUATE-CITIES,* and *RETURN-EQUIPMENT.* Class EVACUATION-OPERATION represents the project to be accomplished - the evacuation mission. The project's three subcomponents correspond to the three phases of an evacuation operation: the movement of evacuation equipment to the location to be evacuated, effecting the evacuation, and returning the evacuation equipment and evacuees to safety.

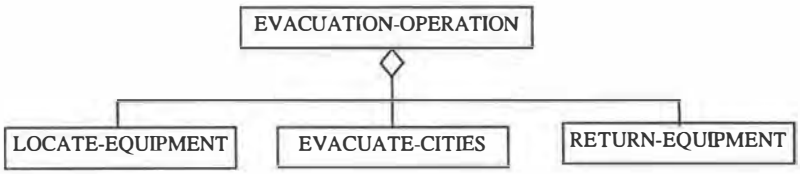


Figure 8-24, First level model of the pacifica domain

Class *LOCATE-EQUIPMENT* is related through the aggregation relationship to the classes *GROUND-TRANSPORTS-IN, AIR-TRANSPORTS-IN* and *CARGO-AIRCRAFT-IN.* This structure is depicted in Figure 8-25 below. This arrangement captures the structure of the domain i.e., the location of equipment is composed of the location of air, ground, and cargo transports. The multiplicity balls specify that an instance of class *LOCATE-EQUIPMENT* may be related to zero or more instances of each of the transport classes.

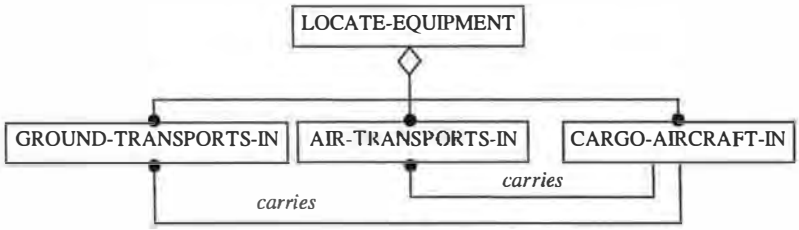


Figure 8-25, Refinement of class LOCATE-EQUIPMENT

The *carries* relationship specifies that an instance of class *CARGO-AIRCRAFT-IN* may be related to zero or more instance of the classes *GROUND-TRANSPORT-IN* and *AIR-TRANSPORT-IN* through a *carries* relationship. The relationship has the semantics that a cargo aircraft carries a number of air and ground transports.

The refinement of class *RETURN-EQUIPMENT* follows the same pattern; hence, the representation is not derived here.

Figure 8-26, below, depicts the refinement of class *EVACUATE-CITIES*. The task of evacuating cities is represented as being related to zero or more cities via the aggregation relationship.

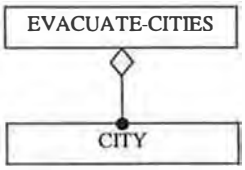


Figure 8-26, Refinement of class EVACUATE-CITIES

With the structure of the domain captured, the second phase of the modelling considers the action which must be attached to each class. The detailed representation and planning applied to classes *LOCATE-EQUIPMENT* and *EVACUATE-CITIES* is described in the two sub sections below.

8.3.1.2 Representation and planning of the *LOCATE-EQUIPMENT* class

This section defines the action required by class *LOCATE-EQUIPMENT* and its subcomponents. Within the domain definition complete, a trace of the integrated architecture solving a specific evacuation mission (Operation-Vanson Figure 8-27) is presented.

After deriving the initial domain structure, the second phase of the encoding demands the attachment of actions to classes. Class *LOCATE-EQUIPMENT* is depicted below with a single abstract action *locate-equipment* attached. This action represents the role of the instances of the class in a model.

```
class LOCATE-EQUIPMENT
    abstract action: must(a, locate-equipment)
```

Class *AIR-TRANSPORT-IN* action's are depicted below. The primitive actions capture the requirement of loading and unloading air transports from the cargo aircraft which carries them to the evacuation location. Class *GROUND-TRANSPORT-IN* contains the same action set.

```
class AIR-TRANSPORT-IN
    carried-by      :  CARGO-AIRCRAFT-IN
    abstract action :  must (a, locate)
    primitive action:  must (a, load onto ?carried-by)
                       must (a, unload from ?carried-by)
```

Class *CARGO-AIRCRAFT-IN* actions are depicted below. The primitive actions capture the requirement of taking off, flying to, and landing at the location to be evacuated (an attribute of class *EVACUATION-OPERATION*).

```
class CARGO-AIRCRAFT-IN
    carries:        AIR-TRANSPORT-IN, GROUND-TRANSPORT-IN
    abstract action :  must(a, locate)
    primitive action:  must (a, take-off ?self.initial-location)
                       must(a, fly-to ?EVACUATION-OPERATION.evac-location)
                       must (a, land-at ?EVACUATION-OPERATION.evac-location)
```

Classes *LOCATE-EQUIPMENT* has the single primitive actions *locate* attached to describe the overall task the class represents.

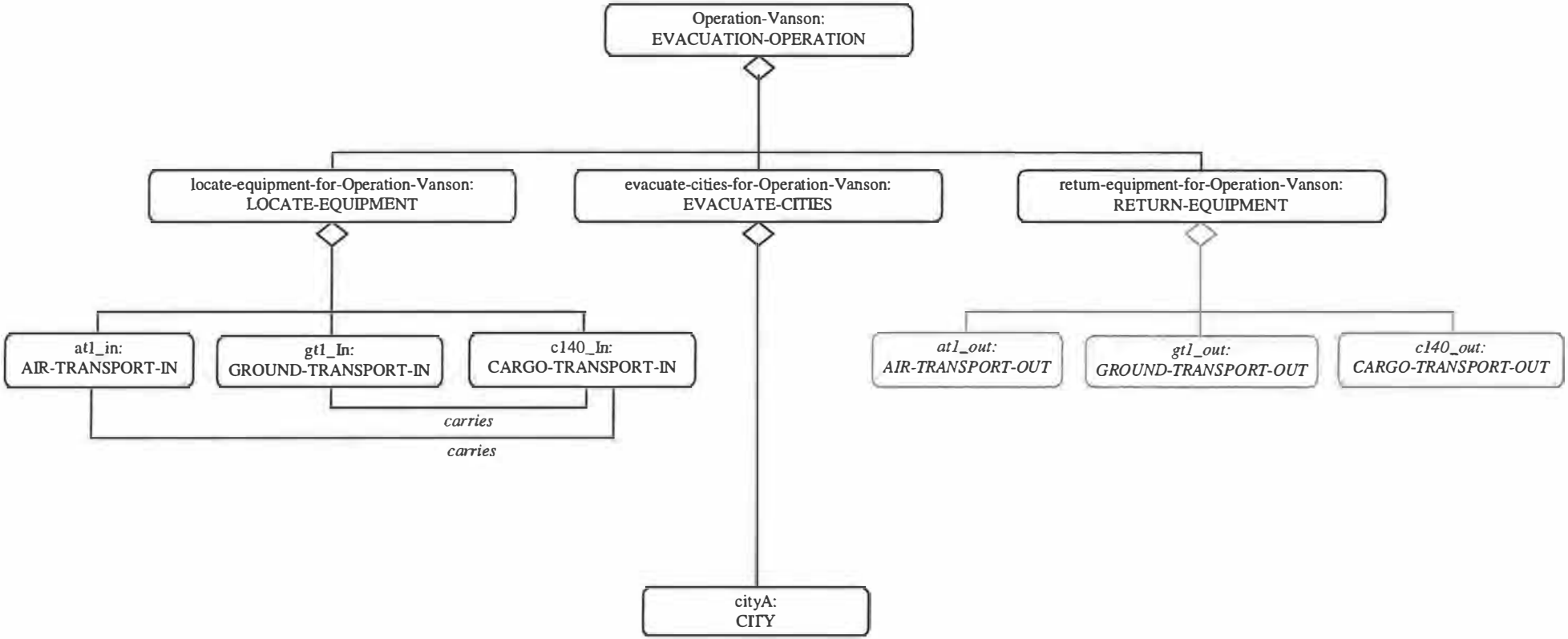


Figure 8-27, Initial instance model for operation Vanson

The third phase of the encoding requires the writing of a *determine-dependency* method for each class. Class *CARGO-AIRCRAFT-IN* requires the following *determine-dependency* method. The method links the instance of action class *TAKE-OFF* associated and instance of the cargo class to be before the instance of the action class *FLY-TO*.

```
class CARGO-AIRCRAFT-IN
    method : determine-dependency
        {
            link-primitive (take-off, fly-to);
            link-primitive (fly-to, land-at);
        };
```

The following *determine-dependency* method is written for classes *AIR-TRANSPORT-IN* and *GROUND-TRANSPORT-IN*. The method links the *load* action of each *transport* to before the *take-off* action of the *cargo-craft* which carries the *transport*. The unloading of each transport is constrained to take place after the cargo aircraft has landed at the destination.

```
class AIR-TRANSPORT-IN
    method : determine-dependency
        {
            link-with-effects (self.load-onto, carried-by, take-off)
            link-with-effects (self.unload-from, carried-by, land-at)
        };
```

A textual representation of the Operation-Vanson mission definition whose parameters are depicted in Figure 8-23 is presented below:

```
instance Operation-Vanson
    instance of class EVACUATION-OPERATION
    evac-location  : pacifica
    sub : locate-equipment-for-Operation-Vanson, evacuate-cities-for-operation-
Vanson, return-equipment-for-Operation-Vanson

instance locate-equipment-for-Operation-Vanson
    instance of class LOCATE-EQUIPMENT
    sub :    : atl_In, gtl_In, In c140_In.
```

The trace of the integrated planner solving the location of equipment for the Vanson mission is depicted below.

```
user> Invoke MBP
---- MBP V4 Invoked ---
one instance of class project = Operation-Vanson
planning for Operation-Vanson
Generate actions(Operation-Vanson)
    o_get_v_d_f(Operation-Vanson.abstract action) return effect
        :because: must(a, effect) directive
    Operation-Vanson is not primitive
    processing subcomponents of Operation-Vanson
    for all subcomponents of Operation-Vanson over set = {locate-equipment-for-operation-
        Vanson, evacuate-cities-for-Operation-Vanson, return-equipment-for-
    Operation-Vanson}
        looping for locate-equipment-for-Operation-Vanson
            Generate actions (locate-equipment-for-Operation-Vanson)
                o_get_v_d_f(locate-equipment-for-Operation-Vanson.abstract action) returns
                locate
                :because: must(a, locate) directive
            locate-equipment-for-Operation-Vanson is not primitive
                for all subcomponents of locate-equipment-for-Operation-Vanson over set =
                { atl_In, gtl_In, cl40_In, }
                looping for atl_In
                    Generate actions(atl_In)
                        o_get_v_d_f(atl_In.abstract-action) returns locate
                        :because: must(a, locate) directive
                    atl_In is primitive
                        o_get_v_d_f(atl_In.primtive-actions) returns {load-onto-cl30, unload-
                                                                    from-cl30}
                        :because: of directives must (a, load-onto ?carried-by) where ?carried-by
                        evaluated to cl30, must(a, unload-from ?carried-by) where ?carried-by
                        evaluated to cl30
                processing of atl_in complete
                looping for gtl_in
                    ...
                processing of gtlin complete
                looping for C150_In
                    Generate actions (C150_In)
                        o_get_v_d_f(Gl50_In.abstract action) returns locate
                        :because: must(a, locate) directive
                    C150_In is primitive
                        o_get_v_d_f(Cl50_In.primitve actions) returns {take-off-from US-
                        Air-base-1, fly-to PACIFICA, land-at PACIFICA}
                        :because: must(a, take-off-from ?self.initial-location) where ?self.initial-
                        location evaluated to US-Air-base-1 ...
                processing of C!50_in complete
            processing of locate-equipment-for-Operation-Vanson complete

-- DEBUG MESSAGE> MBP instructed to IGNORE remaining subcomponents of Operation-
Vanson
```

Figure 8-28, Trace of the MBP component solving operation Vanson

The results of the MBP planning process in Figure 8-28 are presented textually in Figure 8-29 and graphically in Figure 8-30 below.

The textual representation includes the effects attached to each action. The graphical representation permits the abstract action, primitive action, and activity structure to be observed.

```
--- MBP action instances

instance effect1
    instance of class   EFFECT
    object:             Operation-Vanson
    main. Effect:       Operation-Vanson = completed
    sub                 {locate1}
instance locate1
    instance of class   LOCATE-EVACUATION-EQUIPMENT
    object:             locate-equipment-for-Operation-Vanson
    main. effects:      equipment = located
    super:              {effect1}
    sub:                {locate2, locate3, locate4}
instance locate2
    instance of class   LOCATE-AIR-TRANSPORT
    object:             atl_In
    main. effects:      atl_In = located
    super               {locate1}
    sub                 {load-ontoc130-1, unload-from-c130-1}
instance load-onto-c130-1
    instance of class   LOAD-AIR-TRANSPORT-ONTO-CARGO
    object:             atl_In
    main. effects       atl_in = loaded
    super {locate2}
instance unload-from-c130-1
    instance of class   UNLOAD-AIR-TRANSPORT-FROM-CARGO
    object              atl_In
    main. effects       atl_In = unloaded
    super {locate2}
instance locate 4
    instance of class   LOCATE-CARGO-AIRCRAFT
    object              c130
    main. effects       c130 = located
    super               {locate1}
    sub                 {take-off-from-IS-air-base-1, fly-to-Pacifica-1, land-at-Pacifica-
                        1}
instance take-off-from-US-air-base-1-1
    instance of class   CARGO-AIRCRAFT-TAKE-OFF
    object              c130
    main. effects       c130 = airborne, runway US-air-base1 clear
    conditions          runway US-air-base-1 clear
    super               {locate4}
instance fly-to-Pacifica-1
    instance of class   CARGO-FLY-TO
    object              c130
    main. effects       c130 location = pacifica air space
    super               {locate 4}
instance land-at-Pacifica-1
    instance of class   CARGO-LAND-AT
    object:             c130
    main. effects       c130 location = pacifica runway
    conditions          pacifica runway = clear
    super               {locate 4}
```

Figure 8-29, Actions synthesised with conditions and effects

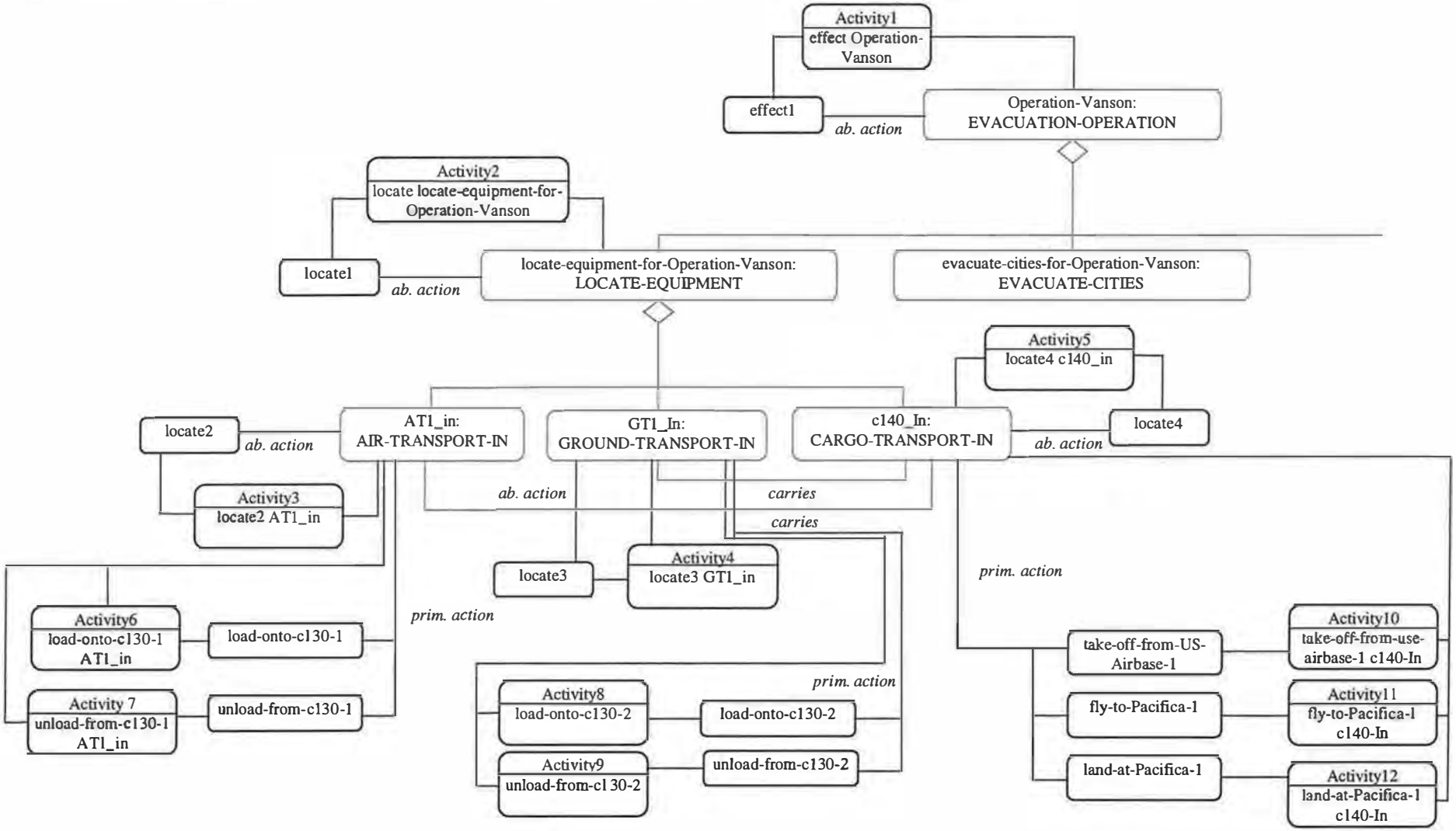


Figure 8-30, Graphical representation of actions and activities synthesised

With action synthesis complete, the MBP invokes the dependency synthesis phase. A trace of this process is presented in Figure 8-31 below.

```
--- MBP generating dependency
    for all objects
        processing object Operation-Vanson
            generate dependency returns null
        processing object locate-equipment-for-Operation-Vanson
            generate dependency returns null
        processing object at1_In
            link-with-effects (self.load-onto, carried-by, take-off) results in
                self.load-onto = load-onto-c130-1,
                carried-by = c140_in
                take-off = take-off-from-US-Airbase1
                linking load-onto-c130-1 before take-off-from-US-airbase-1
            results Acitivty9 dependent upon activity6
            link-with-effects (self.unload-from, carried-by, land-at)
                self.unload-from = unload-from-c130-1
                carried-by = c140-in
                land-at = land-at-Pacifica-1
                linking land-at-Pacifica-1 before unload-from-c130-1
            results Activity 7 Dependent upon Activity11
        processing object gt1_In
            ....
        processing object c140_in
            ...
    end generate dependency
--- Generation of relationship dependency complete
--- No aggregate dependency conditions.
--- MBPV4 complete.
```

Figure 8-31, Dependency synthesis in the pacifica domain

With dependency synthesis complete, the integrated planner initiates the HTN component of the architecture. A trace of this process is presented in Figure 8-32 below.

```
HTN-Planner interface invoked
    generating task
        project =   Operation-Vanson
        project.abstract action = effect1
        generating task network
                ; task effect1-opeation-Vanson
                ;   nodes       1. start
                ;               2. end
                ;               3. action (effect Operation-Vanson)
                ;   orderings   1-->3, 3-->2;
                ; end task ;;; effect1-Operation-Vanson
HTN Planner Invoked
    issues = {action {effect Operation-Vanson}}
    picked issue: action{effect Operation-Vanson}
    Request integrated-architecture-expand {effect Operation-Vanson}
    new-schema-name = effect-Operation-Vanson
    returns schema
                ; schema effect1-opeation-Colubmus
                ;   nodes       1. action( loacate1 locate-equipment-for-operation-
                                Vanson)
                ; only_use_for_effects
                ;   equipment located at 1
                ; end schema;
    issues = action {loactel locate-equipment-for-Operation-Vanson}
    picked issues action {locate1 locate-equipment-for-Operation-Vanson}
    Request integrated-architecture-expand (loate1 locate-equipment-for-operation-
    Vanson}
    new-schema-name = locate1-locate-equipment-for-opeation-colbums
    returns schema
                ; schema locate1-locate-equipment-for-operation-columbus
                ;   nodes       1. action (locate2 at1_In)
                ;               2. action (locate3 gt1_In)
                ;               3. locate4 (c140_in)
                ; only_use_for_effects
                ;   at1_In located at 1
                ;   gt2_In located at 2
                ;   c140_in located at 3
    issues = action {locate2 at1_In }, action{locate3 gt1_In}, action {locate3
    c140_In}
    picked issues action {locate2 at1_In}
    Request integrated-architecture-expand (llocate2 at1_In }
    new-schema-name = Locate2 at1-In
    returns schema
                ; schema late2-at1-In
                ;   nodes       1. action (load-onto-c130-1 at1_In)
                ;               2. action (unload-from-c130 at1_In)
                ;   orderings   1-->2
                ;   only_use_for_effects
                ;     at1_In load at 1
                ;     at1_In unload at 1
                ;   conditions
                ;     supervised at1 loaded at 2 from 1
                ;   encapsulation buster land-at-pacifica-1 c140 at 2
```

Figure 8-32, Trace of the HTN phase of planning operation Vanson

The encoding above successfully synthesises the actions, conditions, ordering constraints, and effects required by the locate equipment phase of operation Vanson

8.3.1.3 Representing and planning for class *EVACUATE-CITIES*

The *EVACUATE-CITIES* class represents the cities to be evacuated. The class is related through the *sub* relation to class *CITY*. Figure 8-33, below, presents an instance of the cities to be evacuated within a specific Pacifica operation.

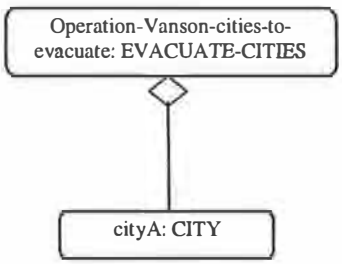


Figure 8-33, Instance diagram of the cities to be evacuated within a Pacifica operation

The actions attached to the classes *EVACUATE-CITIES* and *CITY* are depicted below.

```
class EVACUATE-CITIES
    abstract action : must(a, evacuate-all)
end class

class CITY
    abstract action: must(a, evacuate ?self.no-people-to-evacuate}
    primitive action :null;
end class
```

CITY has no primitive actions defined as the integrated planner produces only the appropriate number of evacuation actions. The assignment of *evacuation equipment* to cities is performed using the traditional HTN approach. The schema resulting from the instance diagram in Figure 8-33 is depicted below.

```
schema evacuate-all-Operation-Vanson-cities-to-evacuate
    expands {evacuate-all Operation-Vanson-cities-to-evacuate}
    nodes       1. action { evacuate 50 cityA}
    end schema.
```

This function demonstrates the strength of the integrated architecture. The assignment of cargo aircraft etc. is performed in the new way. The evaluation of the number of cities to be evacuated takes advantage of the variable node reasoning within the integrated architecture. The precondition achievement search of refining the cities to be evacuated is obtained in the traditional way.

8.4 Automated planning expert perspective

The expert evaluation perspective obtained comments from automated planning experts via conference paper submissions and presentations. The publications relating to this section are (Jarvis and Winstanley 1996a; 1996b). In addition to the successful submissions, papers were submitted to Third International Conference on Automated Planning Systems (AIPS-96) and the 15th National Conference on Artificial Intelligence (AAAI-97). The comments received from these conferences' blind reviews are included within this evaluation.

The automated planning expert evaluation is presented under three headings below: originality, expressiveness concerns, and semantic gap concerns.

8.4.1 Originality

Whilst this thesis has worked to determine the position of the integrated architecture within existing planning theory, the author was concerned that other work in the area of this thesis had not been identified. Comments received from the AAAI-97 blind review summarised the integrated architecture as "*very original*" and that it may "... *lead to interesting application frameworks*".

The originality comments correlate with the discussion within this thesis to reinforce the conclusion that the integrated architecture is a new contribution to planning theory.

8.4.2 Expressiveness argument concerns

Expressiveness argument concerns centre around two related points. First, that a simple domain representation as supported by precondition achievement planning is desirable. Second, that the integrated approach moves much of the complexity of planning away from the domain independent planning engine and into the domain theory.

Precondition achievement planning aims to provide a simple declarative action representation and a powerful domain independent planning algorithm which will identify and order the actions required by any problem solvable with the action set represented. Within this framework, the problem solving capability is implemented within the domain independent planning algorithm. The task of encoding a new domain becomes one of identifying and declaring primitive actions only.

The precondition achievement approach is desirable from both the software engineering and functionality perspectives. First, applying the technology to new domains requires only primitive actions to be identified. Second, the resultant encoding will solve any problem, however unforeseen, which can be solved with a sequence of the actions specified.

Whilst precondition achievement planning has desirable properties, the technology is proving difficult to realise. The sustained effort applied to the technology since the 1960's have failed to address completely the prohibitive search space encountered when attempting to realise precondition achievement planning on realistic problems. Whilst much progress has been made in this area (see Chapter 2), at the time of writing the application of precondition achievement planning has been limited to either small *toy* domains engineered by planning system designers or industrial domains with a small number of actions (of the order of five to ten).

The industrial success of task refinement planning has resulted from the provision of constructs which permit a domain writer to specify action hierarchies and casual structure. These constructs both increase the effort required to write a domain description and may sacrifice completeness. The latter restriction occurs because the planning engine will follow the constraints on condition establishment specified by the domain writer, hence, all possible actions, orderings and variable bindings will not be considered.

The integrated architecture is a logical progression from the task refinement approach. A domain writer may not only specify action hierarchies and causal structure, but also the knowledge from which these structures are generated.

In answer to the expressiveness concerns, precondition achievement planning has not yet reached its industrial potential. Whilst it has desirable properties, if one wishes to implement an automated planning application today, one must consider either a task refinement only planner or an integrated MBP and task refinement planner.

8.4.3 Semantic gap

Planning experts raised two points relating to the semantic gap between application domain knowledge and representation formalism. First, would domain experts prefer to encode their knowledge within the MBP formalism used in the integrated architecture or traditional task refinement schemata? Second, is the isolation of the HTN engine's inference structure from the domain writer simplifying the writing of domain descriptions or simply making the connection more opaque?

Taking the first question, without performing experiments to compare experts writing descriptions in both task refinement and integrated formalisms, it is not possible to provide a definitive answer to this question. However, the planning community provides little assistance in the form of methods for writing domain descriptions. The constructs supported by the integrated architecture map directly to those supported by object-oriented based methodologies. Thus, one may conclude that the integrated architecture is closer to tool supported methodologies than current classical formalisms.

In answer to the second question, the compilation of conditions from domain relationships does isolate the domain writer from the condition type mechanism utilised by the task refinement component of the integrated architecture. This isolation is desirable if the new constructs map closely to the knowledge within applications domains, reducing the need for the domain writer to understand the mechanisms of the planning engine. It is not, however, possible to evaluate this conclusion without further experimentation.

8.5 Industrial planning expert perspective

Construction industry planning experts expressed two concerns: the domain modelling effort, and the ability to update and maintain a completed plan.

The planning experts noted the time and effort demanded by the elicitation phase required to capture a fragment of the knowledge utilised within a relatively small construction project. Extrapolating this effort to all the possible components used within construction projects, the knowledge elicitation overhead would prohibit a single organisation implementing an automated planning system.

If automated planning is to be implemented in the construction industry, an industry wide approach is required to spread the modelling overhead. This issue is expanded in the further work section within Chapter 9.

With the ability to automate the synthesis of plans established, the experts' concern moved from the feasibility of automated planning to the issues of updating and maintenance of plans. A multi million pound construction project would require a full-time planning expert. At a cost of approximately £80,000 per year, the expert is not a significant cost to the project. It is the ability of this expert to identify and cost issues arising from design changes which effect the profitability of a project. For example, a customer may change the type of door used within a hotel building after the project has commenced. Such a change may affect the order in which other activities may completed resulting in a increase in construction cost. The human planner must identify and justify these costs to the customer to a standard where the customer will agree to meet the extra costs or reverse the design change.

The research reported in this thesis has been concerned with plan synthesis. The expert's requirements identify that plan synthesis must be integrated with the complete process under which planning is performed. This issue is expanded in the further work section within Chapter 9.

8.6 Summary and conclusions

The integrated architecture developed in Chapter 7 was motivated by a number of complementary strengths between classical and model-based planning technologies. The strengths and limitations perspective identified the facet(s) of the integrated architecture which address each of these limitations. Exploiting the expressive mechanisms which may be applied to a static domain model, the integrated architecture synthesis the actions, conditions, effects, and ordering constraints required by a specific problem instance. The results of this process are combined and maintained by a traditional task refinement planning system.

Applying the integrated architecture to the Pacifica domain formed the literature evaluation perspective. The perspective demonstrated the generality of the architecture and the benefits of providing a mixed mode integrated - task refinement capability. Whilst planning the location of evacuation equipment within the Pacifica domain, the architecture was able to exploit both the structure of the domain and the expressive inference mechanisms of the integrated architecture. When solving the search based evacuation of outlying cities, the precondition achievement functionality of the task refinement planner was exploited

The automated planning expert evaluation raised two issues. First, it motivated a comparison between the integrated approach and precondition achievement planning. Second, it identified the need for further work to examine experts' encoding preferences.

Taking the first case, precondition achievement planning's aim is to provide the desirable capability of a declarative action representation combined with a domain independent planning algorithm which can solve any problem realisable with a domain's actions. The search space which must be navigated to achieve such a functionality has not yet, however, been adequately addressed. Whilst the integrated architecture requires domain dependant inference routines to be written, the architecture does exploit the domain independent question and answering algorithms supported by classical planning. The writing of domain dependant inference routines is a necessary overhead if one wishes to implement an industrial planning application within existing planning theory.

Taking the second case, it is not possible to definitively answer the question of domain experts' encoding preferences without experimentation. Such a study would require observations of experts encoding their domain knowledge within task refinement and integrated formalisms. However, the constructs supported by the integrated architecture map closely to those supported by current object-oriented methodologies. Thus, the integrated architecture provides a representation which is closer to tool supported methodologies than existing task refinement planners.

In conclusion, the integrated architecture provides a new approach to automated planning which combines the relative strengths of classical and model-based technologies. The compilation of task networks from a domain model permits expressive constructs to be exploited, addressing the expressiveness, redundancy, and semantic distance issues identified in Chapter 4 and Chapter 6. The assembly of task networks into a complete plan permits the powerful condition and effect reasoning mechanisms of task refinement planning to be exploited to ensure a consistent plan with condition and effect constraints established and maintained.

Blank

9. Summary, conclusion, and further work

This is not the end. It is not even the beginning of
the end. But it is, perhaps, the end of the beginning.
Winston Churchill (1874 - 1965)

9.1 Summary

Precondition achievement planning aims to provide a declarative action representation, encapsulating planning knowledge in a domain independent planning algorithm. The computational complexity in providing this function has yet to be addressed; limiting the technology to the research laboratory.

Task refinement planning permits a domain to be structured into a hierarchy of actions with constraints on the methods of establishing and maintaining conditions. Whilst increasing the effort of writing domain descriptions and losing completeness, the technology addresses in part the computational complexities of planning, allowing task refinement planners to be applied to industrial problems.

Model-based planning has been developed in isolation by an effort concerned with implementing industrial planning applications. Whilst the technology has achieved industrial success, its independence from classical planning has prohibited a cross fertilisation of ideas.

Model-based planning may benefit from classical planning's conditions, effects, and associated question and answering function for establishing and maintaining a plan's causal structure. Task refinement planning may benefit from model-based planning's constructs for capturing and reasoning with a domain expert's planning knowledge.

The integrated architecture exploits model-based planning's reasoning to synthesise the actions, conditions, effects, and ordering constraints required by a problem. The results of this reasoning are compiled into task networks for assembly by a task refinement planner into a complete and consistent plan.

Whilst the architecture's applicability is limited to domains structured around the subcomponent relation, in such cases, it provides a more effective technique than either technology applied in isolation.

9.2 Conclusion

A universal domain-independent planning algorithm is a desirable functionality. NASA's autonomous exploration goals are a case-in-point of the most demanding software control applications. NASA plans to send "... *spacecraft where we cannot see, let them search beyond the horizon, accept that we cannot control them while they are there, and rely on them to tell the tale"* (reported in Williams 1996, pp 279). Williams notes that the number of issues that software controlling such spacecraft must entertain are too large to implement explicitly. The control software must be capable of reasoning with the set of actions available to the spacecraft to generate appropriate courses of action in any situation. Precondition achievement planning's aim is to provide this functionality; however, it has yet to fully address the complexity of the problem.

Precondition achievement planning adds actions and ordering constraints to a plan to establish action preconditions and goal conditions with action effects. Task refinement planning allows a domain to be structured into a set of plan fragments which are assembled to form a plan, making domain specific knowledge available to a domain independent algorithm.

This thesis demonstrates that it is not feasible to encode a complex domain into plan fragments. The integrated architecture provides an alternative to the precondition achievement approach of synthesising actions based solely on establishing conditions. By viewing task networks as the result of a planning phase, actions, conditions, effects, and ordering constraints may be synthesised from a central domain model. As this reasoning is in a space where determining the truth of conditions is computationally inexpensive, an expressive formalism which maps closely to a domain expert's knowledge may be used. By exploiting task refinement mechanisms, the results of the compilation phase may be assembled into a complete and consistent plan.

In terms of implementation detail, this thesis describes how task networks can be compiled from a static domain model structured around objects and the subcomponent relationship. In terms of approach, this thesis demonstrates the planning function which can be achieved in the space of a static domain model, and that the results of this reasoning may be combined using classical techniques. Whilst precondition achievement planning is a desirable functionality, it has not yet developed to meet the demands of commercial applications. The integrated architecture provides an industrial architecture for addressing specific planning applications.

9.3 Further work

This section describes research directions motivated by the integrated architecture.

9.3.1 Analysis of the integrated architecture as a problem solving method

Knowledge Engineers have noted that the frame and rule constructs used to implement knowledge based systems do not provide a sufficient abstraction when addressing complex domains. The motivation for the integrated architecture detailed in Chapters 4 and 6 adds weight to the argument that task refinement constructs suffer the same limitation.

Chandrasekaran (1983) advocates the use of domain independent *problem solving methods* (PSM) as a more suitable abstraction. PSMs specify different ways in which a problem solver makes use of inferences from a knowledge base. If a developer has a library of such methods available, they may identify an appropriate method and then use it to structure the knowledge base development. Protété II (Eriksson et. al. 1995) is a meta-tool for supporting a developer in taking their analysis of a function a knowledge based system is to discharge and matching that analysis with an appropriate method from a library of PSMs. Protété II assists in the case of several methods being appropriate to achieve a task by indicating the conditions under which each is appropriate. Selection criteria range from the availability of expertise to computation time and space requirements.

Further work is suggested to consider the integrated architecture as a problem solving method alongside the *propose critique modify* classification of classical only planners developed by Valente (Valente 1995). This work requires the formalisation and development of criteria leading to the selection between methods and their implementation within a tool similar to Protété II.

A useful extension would be the study of construction planning and planning domains in general to test the generality of the integrated architecture as a planning problem solving method. This work could continue the construction of the KADS model set presented in chapter 5 to define the task and inference layers of the expertise model. This analysis may identify new structures which can be mapped onto classical planners through a process similar to the compilation phase of the MBP. A major contribution would be the identification of the facets which lead to each methods applicability in preference to others.

This is an important further development of the integrated architecture. The distance between human knowledge and a task refinement formalism is at present to great. By providing structured domain encodings methods akin too those offered by the architecture the domain modeller is provided with constructs which map more closely to the domain under consideration.

9.3.2 Tool support for model construction

It is well understood that communication between a knowledge engineer and experts is difficult. The knowledge engineer initially lacks the domain knowledge to ask optimal questions and the expert often finds it difficult to inspect and communicate their own knowledge. These issues were encountered during the engineering of the construction domain described in Chapter 5

To address this issue it has been suggested that experts play a greater role in the encoding of knowledge directly into computers, hence, removing the need in part to communicate with knowledge engineers. The OPAL tool (Musen et al. 1987) developed to aid expert knowledge editing for the ONCOCIN (Shortliffe et. al. 1981) cancer treatment expert system is an example of this work. In OPAL experts are not expected to understand the constructs used to internally encode knowledge. Instead the user is provide with a tool based on the structure of the cancer treatment domain itself. The expert may the edited directly the system's knowledge base.

Two related tools inspired by OPAL are described below as suggested further work to support the development of knowledge bases for the integrated architecture.

Domain structuring tool

The MBP modelling constructs are structured according the well defined rules. For example, a model may contain only one *PROJECT* class and only classes of type *PRIMITIVE* may have both *abstract* and *primitive* actions associated. A domain structuring tool may be implemented to both prompt and assist a knowledge engineer when encoding a domain within the MBP constructs.

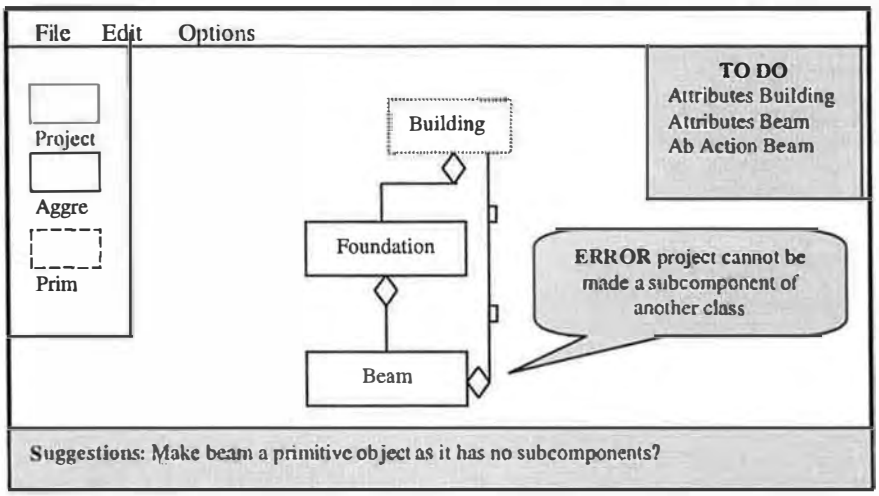


Figure 9-1, Suggested structuring tool interface

Figure 9-1 depicts a suggested interface to a structuring tool and demonstrates a number of the possible support features. On the left of the figure is a palette of available modelling constructs. The knowledge engineer may select from the palette and create the model in the centre of the figure. The right side depicts a *to-do* list indicating the detail the domain modeller must complete. In the example defining the attributes of the classes and the abstract actions of class *BEAM* are shown as outstanding. At the bottom of the figure a suggestion bar offers advice to the domain modeller. Within the model a sample error message is depicted indicating the modeller has violated a modelling constraint. In this instance, the modeller has linked the *PROJECT* class as a subcomponent of another class, thus, invalidating the model.

Whilst the tool offers support for constructing a valid model, its constructs are oriented towards the underlying representation and not the domain under consideration. The second *domain refinement tool* is suggested below for use by domain experts after a knowledge engineer has initially structured a domain.

Domain refinement tool

The proposed domain-structuring tool supports the knowledge engineer in generating the initiation structure of an application domain. The domain refinement tool is suggested to allow domain experts to enter detailed domain knowledge. This second tool is motivated by the concepts introduced in OPAL.

There are two stages envisaged to developing such a tool for a specific domain. First, identifying the mapping from an "experts" domain model to the MBP constructs. For example, identifying how domain experts model the association between component classes and actions would lead to the development of capture tool permitting experts to directly enter knowledge of this type. This process may be repeated for each of the knowledge types required by the MBP. The initial domain structure composed with the structuring tool may aid the process. Specifically, defining the high level classes within a domain will aid the understanding of the domain vocabulary and specify the classes which may be refined further and the default attributes, relationships, and knowledge which will be required by new classes.

The suggested second stage defines how the individual knowledge capture tools defined in the first stage may be combined into a complete tool. Example issues include the order in which questions are asked and the level of freedom the expert is permitted in selecting knowledge types to enter. An important consideration is the interleaving between planning and knowledge capture. Once a critical mass of knowledge has been encoded, planning may then proceed with the system prompting experts for knowledge about new, possibly more specialised and therefore less common, entities.

The result is envisaged as a support tool, derived by the underlying MBP constructs and a knowledge engineer's structuring of a domain that allows domain experts to enter detailed domain knowledge. The need for such a tool was highlighted by the construction industry expert evaluation in Chapter 8. Experts were concerned at the cost of capturing the vast quantify of knowledge within the industry. Tool support would allow a number of experts, possibly from different organisations, to collectively build a domain model with each contributing knowledge from their specific areas of expertise.

9.3.3 Explanation

Current explanation tools may exploit the structures generated by task refinement planners to explain the rational behind a plan. Further work is suggested to make available the MBP structures to such tools.

The structures generated by task refinement planners are centred on condition establishment from action effects (GOST), expansion history of an action (i.e. the higher level tasks from which it was included), and the filter condition selection mechanism. From this set, it is possible to present a plan rational in the form below:

- Action *install bolt-sink-to-wall* was included as part of the *plumbing* task.
- The *Plumbing-1* method was used to achieve the *plumbing* task as the methods *only_use_if* condition *EEC-Standard-to-plumbing* equalled *applied.*
- Action *move plant from basement* was included to protect the precondition of action *pour basement floor* named *basement-floor-area clear.*
- Action *move plant from basement* was ordered after action *dig-basement-structure* as the action is required by the *construct-basement-infrastructure* task.

MBP utilises rule-based reasoning for determining the need for actions and relationships between components and rule-based reasoning for ordering constraints between actions. These mechanisms encode and reason with domain expert knowledge. With the careful recording of the reasoning processes it would be possible to argument the task refinement explanation with this knowledge.

This suggested direction would require the study of existing techniques for exploiting the results of rule-based reasoning for explanation.

9.3.4 Continued use of industrial planning applications within planning research

The Defence Advanced Research Project Agency / Rome Laboratory Planning Initiative (ARPI) is focusing automated planning research on the demands of industrial planning applications. This work is examining and addressing the issues of why plans are created, how they areas used, updated, and maintained.

The collaboration with an industrial organisation as part of the research reported in this thesis highlighted this integration with an organisation's business process as an essential attribute of industrial automated planning applications. Automated planning research has developed powerful plan synthesis and plan representation technologies. Integration of these technologies with industrial planning tools (e.g. Microsoft Project), databases, and tools (e.g. AutoCAD) is essential for their acceptance into industry. The Optimum-AIV system (Tate 1996c) is an example of work of this type. The system integrates technology based upon the O-Plan system with Matra Marconi Space's planning process for production of vehicle equipment bays for the European Space Agencies' Arian-4 launcher.

9.3.5 Detailed issues

9.3.5.1 Model-based planning algorithm

The model-based planning algorithm completes action synthesis before assessing dependency constraints. In each process, the model is traversed in a depth first path. Actions must be synthesised before dependency constraints are considered as the constraints are placed between actions. If an object's dependency is assessed before the action synthesis process is complete, the actions of the objects upon which the object is dependent are not guaranteed to have been generated.

The depth first transversal of the model ensures that each object is processed. In the current implementation any transversal order is permitted - providing the constraint of processing each object in the model is maintained.

Further research should consider the potential for a non sequential process. For example, there may be benefits from assessing all the actions and dependency constraints at a modelling level before moving to the next. The lower modelling level may then consider the actions and constraints placed at the higher level. Such an approach would allow rules of the form "if self.supercomponent. abstract-action is dependent upon an action of another type, then include action A". This approach permits actions and ordering constraints to be inferred based on other actions and orderings within a plan without considering the complexities of partial-order planning.

9.3.6 Resources

The model-based paradigm associates resources with actions through a similar mechanism as actions to objects. The development of the integrated architecture simplified the issues to be addressed by ignoring the issue of resources.

Further work should consider a new phase to the model-based planning algorithm for the association of resources with actions, and the integration of these structures with the task network compilation algorithm.

10. References

Anderson. S., & Cohen. P., 1996, On-line planning simulation, In Proceedings of The Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, AAAI Press, ISBN 0-929280-97-0.

Andrews. S., Kettler. B., Erol. K., & Hendler. J., 1995, UM Translog: A Planning Domain for the Development and Benchmarking of Planning Systems, Technical report CS TR-3487, Department of Computer Science, University of Maryland, Maryland, USA.

Allen. J., Hendler. J., & Tate. A., 1990 (eds.) Readings in Planning, Morgan-Kaufman Publishers, Palo Alto, California, USA, ISBN 1-55860-130-9

Baader. F., 1990, A formal definition for expressive power of knowledge representation languages. In Proceedings of the 9th European Conference on Artificial Intelligence, Stockholm, Sweden.

Barber. T., Marshall. G., & Boardman. J., 1987, A Philosophy and Architecture for a Rule-Based Frame System: RBFS, *International Journal of Artificial Intelligence in Engineering.*

Barrett. A., Golden. K., Penberthy. S., & Weld. D., 1994, UCPOP User's Manual (Version 2.0), Technical Report 93-09-06, Department of Computer Science and Engineering, University of Washington, Seattle, USA.

Barrett. A., & Weld. D., 1992, UCPOP: A Sound, Complete, Partial Order Planner for ADL, In Proceedings of KR92, 103-114, Cambridge, MA.

Barrett. A., & Weld. D., 1994a, Task-Decomposition via Plan Parsing, In Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94), Seattle, USA

Barrett. A., & Weld. D., 1994b, Partial-order planning: Evaluating possible efficiency gains, *Artificial Intelligence*, vol. 67 no. 1, pp 71-112.

Barros. L., Valente. A., & Benjamins. R., 1996, Modelling Planning Tasks, In Proceedings of The Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, AAAI Press, ISBN 0-929280-97-0.

Beetz. M., & McDermott. D., 1996, Local Planning of Ongoing Activities, In Proceedings of The Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, AAAI Press, ISBN 0-929280-97-0.

Bell. C., & Tate. A., 1985, Using Temporal constraints to Restrict Search in a Planner., Technical report, AIAI-TR-5, Artificial Intelligence Applications Institute, University of Edinburgh, UK.

Benjamins. R., Barros. L., & Valente. A., 1996, Constructing Planners Through Problem Solving Methods, In Proceedings of the 10th Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW 96), Gains. B & Musen. M. editors, Banff.

Blythe. J., 1996, Event-Based Decompositions for Reasoning about External Change in Planners, In Proceedings of The Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, AAAI Press, ISBN 0-929280-97-0.

Booch. G., 1991, Object-oriented Design with Applications, The Benjamin/ Cummings Publishing Company, California, USA, ISBN 0-8053-0091-0

Brooks. R., 1985, A Robust Layered Control System for a Mobile Robot, A.I. Memo No. 864, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Massachusetts, USA.

Brooks. R., 1991, Intelligence Without Reason, A.I. Memo No.1293, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Massachusetts, USA.

Brooks. R., 1991, Intelligence without representation, *Artificial Intelligence*, 47, pp 139-160.

Chandrasekaran. B., 1983, Towards a taxonomy of problem-solving types, *AI-Magazine,* Vol. 4, part 4, pp 9-17.

Chapman. D, 1987, Planning for conjunctive goals, *Artificial Intelligence*, 32, pp 333-378.

Chen. P., 1976, The entity relationship model - toward a unified view of data, TODS 1:1, 1976.

Chien. S., 1996, Static and Completion Analysis for Planning Knowledge Base Development and Verification. In Proceedings of The Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, AAAI Press, ISBN 0-929280-97-0.

Coddington. A., & Aylett. R., 1996, Plan generation for multiple autonomous agents: an evaluation, In Proceedings of the 15th Workshop of the UK Planning and Scheduling Special Interest Group, Liverpool John Moores University, Liverpool, UK.

Collins. G., & Pryor. L., 1992, Achieving the functionality of filter conditions in a partial order planner, In Proceedings of the 10th National conference on Artificial Intelligence (AAAI-92), San Jose, California, USA, pp 375-380.

Collins. G., & Pryor. L., 1994, On the misuse of filter conditions: a critical analysis, in Backstrom. C., & Sandewall. E., eds., Current Trends in AI Planning, IOS press, pp 105-116, Netherlands, ISBN 90 5199 153 3.

Cook. S., & Daniels. J., 1994, Designing Object System: Object-Oriented Modelling with Syntropy, Prentice Hall International, Hertfordshire, UK, ISBN 0-13-203860-9

Currie. K., & Tate. A., 1991, O-Plan: the Open Planning Architecture, *Artificial Intelligence*, vol. 51, part 1, North-Holland.

Dalton. J., 1993, "A simple house-building domain with resources, a.k.a. house 5", [available from] *http://www.aiai.ed.ac.uk/~oplan/demonstrations* [accessed 2nd April 1997].

DeMarco. T., 1982, Controlling Software Projects, Yourdon Press, New York, USA.

Drabble. B., & Kirby. R., 1991, Associating A.I. Planner Entities with an underlying Time Point Network, In Proceedings of the 1st European Workshop on Planning, Sankt Augustin, Germany.

Drabble. B., & Tate. A., 1994, The Use of Optimistic and Pessimistic Resource Profiles to Inform Search in an Activity Based Planner, In Proceedings of the Second International Conference on Artificial Intelligence Planning Systems, Chicago, USA.

Drabble. B., Gil. Y., & Tate. A., 1995, Yes, But why is that plan better ?, In Proceedings of the International Conference on Artificial Intelligence in the Petroleum Industry, Lillehammer, Norway.

Drummond. M., 1994, On Precondition Achievement and the Computational Economic of Automatic Planning, in Backstrom. C., & Sandewall. E., eds., Current Trends in AI Planning, IOS press, pp 6-13, Netherlands, ISBN 90 5199 153 3.

Erol. K., Hendler. J., & Nau. D., 1993, Semantics for Hierarchical Task-Network Planning, Technical Report, CS-TR-3239, Computer Science Department, University of Maryland, Maryland, USA.

Erol. K., Hendler. J., & Nau. D., 1994a, UMCP: A Sound and Complete Procedure for Hierarchical Task-Network Planning, In Proceedings of the Second International conference on Artificial Intelligence Planning Systems, Chicago, USA.

Erol. K., Hendler. J., & Nau. D., 1994b, Task Refinement Planning: Complexity and Expressivity, In Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94), Seattle, USA.

Erol. K., 1995, Hierarchical Task Network Planning: Formalisation, Analysis, and Implementation, PhD Thesis, University of Maryland, USA.

Eriksson . H., Shahar. Y., Tu. S., Puerta. A., Musen. M., 1995, Task modelling with reusable problem-solving methods, *Artificial Intelligence,* 79, pp 293-326, Elsevier.

Etzioni. O., 1990, Why PRODIGY/EBL Works, in proceedings of the 8th National Conference on Artificial Intelligence (AAAI-90), Boston, Massachusetts, USA.

Etzioni. O., 1993, Intelligence without Robots (A Reply to Brooks), *AI Magazine*, Winter, 1993.

Feigenbaum, E., 1980, Knowledge Engineering: The Applied Side of Artificial Intelligence., Memo HPP-80-21, Artificial Intelligence Laboratory, Stanford University, California, USA.

Ferguson. G., Allen. J., & Miller. B., 1996, TRAINS-95, Towards a Mixed-Initiative Planning Assistant, In Proceedings of The Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, AAAI Press, ISBN 0-929280-97-0.

Fikes. R., & Nilsson. N., 1971, STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, *Artificial Intelligence*, Vol. 5, No. 2, North-Holland publishing Co., Amsterdam, Netherlands.

Fowler. N., Cross. S., Garvey. T., & Hoffman. M., 1996, The ARPA-Rome Laboratory Knowledge-Based Planning and Scheduling Initiative (ARPI), Ed. Tate. A., Advanced Planning Technology, Technological Achievements of the APRA/Rome Laboratory Planning Initiative, AAAI Press, California, USA, ISBN 0-929280-98-9

Fox. M., & Long. D., 1995, Hierarchical Planning using Abstraction, In Proceedings of the ICC on Control Theory and Applications, IEE.

Fox. M., & Long. D., 1996, An Efficient Algorithm for Managing Partial Orders in Planning., In Proceedings of the 15th Workshop of the UK Planning and

Scheduling Special Interest Group, Liverpool John Moores University, Liverpool, UK. Vol. 2, pp 160 - 170.

Fuchs. F., 1996, Multi-Agent Collaboration in Competitive Scenarios, In Proceedings of the 15th Workshop of the UK Planning and Scheduling Special Interest Group, Liverpool John Moores University, Liverpool, UK.

Georgeff. M., 1987, Planning, *The Annual Review of Computer Science*, Vol. 2., Annual Reviews Inc.

Georgeff. M., & Ingrand. F, 1989, Decision making in an embedded reasoning system, In Proceedings of the 11th International Joint Conference on Artificial Intelligence.

Ginsberg. M., 1989, Universal planning: an (almost) Universally bad idea, *AI magazine*, part 10, pp 40-44.

Ginsberg. M., 1993, Essentials of Artificial Intelligence, Morgan Kaufmann publishers, San Francisco, California, USA. ISBN 1-55860-334-4.

Golden. K., Etzioni. O., & Weld. D., 1994, Omnipotence without omniscience: Sensor management in planning, In Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94), Seattle, USA.

Goodwin. R., 1996, Using Loops in Decision-Theoretic Refinement Planners, In Proceedings of The Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, AAAI Press, ISBN 0-929280-97-0.

Haddaway. P, & Hanks. S., 1992, Representations for Decision Theoretic Planning: Utility Functions for Deadline Goals, In Proceeding of the 3rd International Conference on the Principles of Knowledge Representation and Reasoning, pp 71-82, California, USA, Morgan Kaufman.

Jarvis . P., & Winstanley. G., 1996a, Dynamically assessed and reasoned task (DART) networks, in proceedings of the Sixteenth Annual Technical Conference of the British Computer Society Specialist Group on Expert Systems, Cambridge, UK, ISBN 1 899621 15 6

Jarvis. P., & Winstanley. G., 1996b, Objects and Objectives: the merging of object and planning technologies, in proceedings of the 15th Workshop of the UK Planning and Scheduling Special Interest Group, Liverpool, UK.

Joslin. D., & Pollack. M., 1994, Least-Cost Flaw Repair: A Plan Refinement Strategy for Partial-Order Planning, In Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94), USA.

Joslin. D., & Pollack. M., 1995, Passive and Active Decision Postponement in Plan Generation, in proceedings of the 13th European Workshop on Planning, Assisi, Italy.

KADS Consortium, 1997, Rationale of the KADS - II project approach, available from http://www.swi.psy.uva.nl/projects/ComonKADS/ description/subsection3.1.1.html [accessed 15th June 1997]

Kambhampati. S., 1994, Comparing Partial Order Planning and Task Reduction Planning: A preliminary report, In Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94), Seattle, USA, working notes of the workshop on Comparative Analysis of Planning Systems.

Kambhampati. S., 1995, A Comparative Analysis of Partial Order Planning and Task Reduction Planning, *SIGART Bulletin*, Vol. 6, No.1

Kambhampati . S., 1996, Refinement Planning: Status and Prospects, In Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-96), Portland, Oregon, USA.

Kartam. N., Levitt. R., & Wilkins. D., 1991, Extending Artificial Intelligence Techniques for Hierarchical Planning, *ASCE Journal of Computing in Civil Engineering.*

Kingston. J, Shadbolt. N., & Tate. A., 1996, CommonKADS Models for Knowledge Based Planning, In Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-96), Portland, Oregon, USA.

Kitchin. D., & McCluskey. T., 1996, Object-centred planning, In Proceedings of the 15th Workshop of the UK Planning and Scheduling Special Interest Group, Liverpool John Moores University, pp198-210.

Knoblock. C., 1996, Building a Planner for Information Gathering: A Report from the Trenches, In Proceedings of The Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, AAAI Press, ISBN 0-929280-97-0.

Kushmerick. N., Hanks. S., Weld. D, 1995, An algorithm for probabilistic planning, *Artificial intelligence,* 76, pp 239-286

Luger. G., & Stubblefiled. W., 1993, Artificial Intelligence, Structures and Strategies for Complex Problem Solving, 2nd edition, The Benjamin / Cummings Publishing Company Inc., California, USA, ISBN 0-8053-4780-1

Manna. Z., & Waldinger. R., 1974, Knowledge and reasoning in program synthesis, *Artificial Intelligence*, Vol. 6, part. 2, 175-208.

Marshall. G., Boardman. J., & Murray. P., 1987, Project formulation and bidding in the flight simulation industry using Knowledge-based techniques, In Proceedings of the Royal Aeronautical Society International Conference on the Acquisition and Use of Flight Simulation Technology in Aviation Training, 1987.

Marshall. G., 1988, PIPPA: The professional intelligent project planning assistant, PhD thesis, The University of Brighton.

McAllester. D., & Rosenblitt. D., 1991, Systematic non-linear planning. In Proceeding of the 9th National Conference on Artificial Intelligence, Anaheim, California, USA. pp 634-639.

McCarthy. J., & Hayes. P., 1969, Some Philosophical Problems From the Standpoint of Artificial Intelligence, *Machine intelligence 4*, Edinburgh University Press, Scotland, UK.

McCluskey. T., & Porteous. J., 1995, On Extracting Goal Structure from Planning Domain Specifications, In Proceedings of the 14th Workshop of the UK Planning and Scheduling Special Interest Group, Wivenhoe House Conference Centre, Essex University, 22 - 23rd November.

McCluskey. T., & Porteous. J., 1996a, Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency, Technical Report RR9606, School of Computing and Maths, University of Huddersfield.

McCluskey. T & Porteous. J., 1996b, Planning Speed-up via Domain Model Compilation., Ghallab. M., & Milani. A. (eds.), New Directions in AI Planning, pp 233-244, IOS Press.

McCluskey. T., Kitchin. D., & Porteous. J., 1996, Object-Centred Planning: Lifting Classical Planning from the Literal level to the Object Level. In Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence.

McDermott. D., 1991, Regression Planning, *International Journal of Intelligent Systems*, vol. 6, pp 357-416.

McDermott. D., 1992, Transformational Planning of Reactive Behaviour, Technical Report, CSD-RR-941, Yale University, Department of Computer Science, USA.

Minton. S., Bresina, J., & Drummond. M.,1991, Commitment strategies in planning: a comparative analysis, In Proceedings of the International Joint Conference on Artificial Intelligence (IJACI-91), pp259-265

Minton. S., Knoblock. C, Kuokka D, Gill. Y., Joseph. R., & Carbonell. J, 1989, PRODIGY 2.0: the manual and tutorial, technical report, CMU-CS -89-146, School of Computer Science, Carnegie Mellon University, USA.

Minton. S., Bresina. J., & Drummond. M., 1994, Total-Order and Partial-Order Planning: A Comparative Analysis, *Journal of Artificial Intelligence Research*, vol. 2, pp 227 - 262.

Musen. M., Fagan. L, Combs. D., & Shortliffe. E., 1987, Use of a domain model to drive an interactive knowledge-editing tool, *International Journal of Man-Machine Studies,* Vol 26, pp 105-121.

Nau. D., Gupta. S., & Regli. W., 1995, AI Planning Versus Manufacturing-Operation Planning: A Case Study, IJCAI-95.

Newell. A., & Simon. H., 1961, GPS, A Program That Simulates Human Thought, *Lernende Automaten*, Munich, Germany.

Newell. A., Simon. H., 1976, Computer science as empirical enquiry: symbols and search., *Communications of the ACM*, 19:3:113-126

Newell. A., 1982, The Knowledge Level., *Artificial Intelligence*, 18, pp 87-127

O-Plan Team, 1994, O-Plan2 ver 2.2 Architecture Guide ver 2.2, Technical Report, Artificial Intelligence Applications Institute, University of Edinburgh, Scotland.

Parsaye, K. & Chignell. M., 1988, Expert Systems for Experts, John Wiley, Canada, ISBN 0-471-60721-5.

Pednault. E., 1988, Synthesising Plans that Contain Actions with Context-Dependent Effects, *Computational Intelligence*, vol. 4, part 4, pp 356-372.

Pednault. E., 1989, Adl: Exploring the middle ground between STRIPS and the Situation Calculus, In Proceedings of the First Knowledge Representation Conference, San Mateo, California, USA, Morgan Kaufman.

Penberthy. J., & Weld. D., 1992, UCPOP: A sound, complete, partial order planner for ADL, In Proceedings of the 3rd international conference on principles of knowledge representation and reasoning, pp 103-104.

Poet. M. & Smith. D., 1992, Conditional Non-linear Planning, In Proceedings of the First International Conference on Artificial Intelligence Planning Systems Los Altos, CA, Morgan Kaufmann, p 189 197

Poet. M., & Smith. D., 1993, Threat-Removal Strategies for Partial-Order Planning, In Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-93), Washington DC, USA. pp 492-499.

Pratt. I., 1994, Artificial Intelligence, Macmillian Press, London, UK, ISBN 0-333-59755-9

Reece. G., Tate. A., Brown. D., Hoffman. M., & Burnard. R., 1993, The PRECiS Environment, In Proceedings of the 11th National conference on Artificial Intelligence (AAAI-93), Washington, D.C, USA, ARPA-RL planning initiative workshop.

Rumbaugh. J., Blaha. M., Premerlani. W., Eddy. F., Lorensen. W., 1991, Object-Oriented Modelling and Design, Prentice-Hall International, New Jersey, USA. ISBN 0-13-630054-5.

Sacerdoti. E., 1975a, Planning in a Hierarchy of Abstraction Spaces, vol. 5, no 2, *Artificial Intelligence*, North Holland Publishers, Netherlands.

Sacerdoti. E., 1975b, The Non-linear Nature of Plans, In Proceedings of the Fourth International Joint Conference on Artificial Intelligence, pp 206-214, California, USA, International Joint Conference of Artificial Intelligence.

Schreiber. G, 1992, The KADS approach to knowledge engineering, editorial special issue, *Knowledge Acquisition,* vol. 4, part 1, pp 1-4.

Schreiber. G., Wielinga. B., & Breuker. J., 1993, KADS A Principled Approach to Knowledge-Based system development, Academic Press, London, UK, ISBN 0-12-629040-7.

Searle. J., 1980, Minds, brains and programs. *The Behavioural and Brain Science*, 3, pp 417-424

Shortliffe. E., Scott. A., Bischoff. M., Van Melle. W., & Jacobs. C., 1981, ONCOCIN: An expert system for oncology protocol management, In Proceedings of the Seventh International Joint conference on Artificial Intelligence, Vancouver, British Columbia, pp 876-881

Steel. S., 1996, Deductive actions, and their application to plan construction: an outline, In Proceedings of the 15th Workshop of the UK Planning and Scheduling Special Interest Group, Liverpool John Moores University, Liverpool, UK.

Stefik. M., 1981, Planning with Constraints - MOLGEN: Part I, *Artificial Intelligence*, vol. 16, No.2, North-Holland publishing co.

Stillman. J., & Bonsissone. P., 1996, Technology Development in the ARPA/RL Planning Initiative, ed. Tate. A., Advanced Planning Technology, Technological Achievements of the ARPA/Rome Laboratory Planning Initiative, AAAI press, California, USA, ISBN 0-929280-98-9

Sussman. G., 1974, The Virtuous Nature of Bugs, In Proceedings of the First Conference of the Society for the Study of AI and Simulation Behaviour, Sussex University, Brighton, UK,

Swartout. W., & Gil. Y., 1996, EXPECT: A user-centred environment for the development and adaptation of knowledge-based planning aids. In Tate. A. (ed.), Advanced Planning Technology: Technological Achievements of the ARPA/ Rome Laboratory Planning Initiative, AAAI press, USA, ISBN 0-929280-98-9.

Tate. A., 1976, Project Planning Using A Hierarchic Non-linear Planner, Department of Artificial Intelligence Research Report No. 25, Artificial Intelligence Library, University of Edinburgh, 80 South Bridge, Edinburgh, EH1 1HN, Scotland, UK.

Tate. A., 1977, Generating Project Networks, In Proceedings of the International Joint Conferences on Artificial Intelligence, pp 888-893.

Tate. A., Hendler. J.,& Drummond. M., 1990, A Review of AI Planning Techniques, in Allen. J., Hendler. J., Tate. A., (eds.), Readings in Planning, Morgan Kaufmann Publishers Inc., California, USA, ISBN 1-55860-130-9, pp 26 - 49.

Tate. A., 1993a, Authority Management - Coordination between Task Assignment, Planning and Execution, In Proceedings of the International Joint Conference on Artificial Intelligence workshop on Knowledge-based Production Planning, Scheduling and Control.

Tate. A., 1993b, The Emergence of "Standard" Planning and Scheduling system components -Open planning and Scheduling Architectures", Second European Worksop on planning, Vadstena, Sweden, Published in Backstrom. C., & Sandewall. E., eds., Current Trends in AI Planning, IOS press, pp 6-13, Netherlands, ISBN 90 5199 153 3.

Tate. A, 1994, Mixed Initiative Planning in O-Plan2, In Proceedings of ARPI Workshop, Tucson, Arizona, Morgan Kaufmann, USA

Tate. A., Drabble. B., & Dalton. J., 1994a, The Use of Condition Types to Restrict Search in an AI Planner, In Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94), Seattle, USA.

Tate. A., Drabble. B., & Dalton. J., 1994b, O-Plan Version 2.2 Task Formalism Manual, O-Plan Project documentation, AIAI, University of Edinburgh.

Tate. A., Drabble. B., Kirby. R., 1994b, O-Plan2: an Open Architecture for Command, Planning and Control, in Intelligent Scheduling, (eds. M. Zweben and M .S .Fox) Morgan Kaufmann, USA.

Tate. A., 1995, Representing plans as a Set of Constraints - the <I-N-OVA> Model - A framework for Comparative Analysis, *ACM SIGART Bulletin,* vol. 6, No. 1, January.

Tate. A., 1996, Representing plans as a Set of Constraints - the <I-N-OVA> Model, In Proceedings of The Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, AAAI Press, ISBN 0-929280-97-0.

Tate. A., 1996b, Editor, Advance Planning Technology, Technological Achievements of the APRA/Rome Laboratory Planning Initiative, AAAI Press, California, USA, ISBN 0-929280-98-9

Tate. A., 1996c, Responsive planning and scheduling using AI planning techniques - Optimum-AIV, *IEEE Expert Intelligent Systems & Their Applications,* Vol. 11, No. 6, December, pp 4 - 12, IEEE.

Tate. A., 1997, Mixed Initiative Interaction in O-Plan, In Proceedings of Computational Models for Mixed Initiative Interactions, AAAI Spring Symposium, USA.

Tsuneto. R., Erol. K., Hendler. J., & Nau. D., 1996, Commitment Strategies in Hierarchical Task Network Planning, In Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-96), USA.

Turban. E., 1992, Expert Systems and Applied Artificial Intelligence, Macmillian Publishing Company, New York, USA, ISBN 0-02-421665-8

Valente. A., 1995, Knowledge level analysis of planning systems, *SIGART Bulletin*, vol. 6, no 1, pp 33-41.

Veloso. M., 1992, Learning by analogical reasoning in general problem solving, PhD thesis, Carnegie Mellon University, School of Computer Science, USA.

Vere. S., 1983, Planning in Time: Windows and Durations for Activities and Goals. *Pattern Analysis and Machine intelligence*, vol. 5, pp 246-247, IEEE.

Waldinger. R., 1977, Achieving Several Goals Simultaneously, *Machine Intelligence 8,* Ellis Horwood Limited, Chichester, UK.

Wang. X., 1994, Learning by observation and practice: an incremental approach for planning operator acquisition. In Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94), Seattle, USA

Wang. X, 1996, Planning While Learning Operators, In Proceedings of The Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, AAAI Press, ISBN 0-929280-97-0.

Weld. D., 1994, An Introduction to Least Commitment Planning, *Artificial Intelligence Magazine*, Winter, pp 27 -61, USA.

Weld. D., & Etzioni. O., 1994, The First Law of Robotics (a call to arms), In Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94), Seattle, USA.

Weld. D., 1996, Planning-Based Control of Software Agents, In Proceedings of The Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, AAAI Press, ISBN 0-929280-97-0.

Wielinga. B., Schreiber. G., & Breuker. J., 1993, Modelling Expertise, in KADS a principled Approach to Knowledge-Based System Development, Schreiber. G., Wielinga. B., Breuker. J., (eds.) Academic Press, London, UK, ISBN-0-12-629040-7, pp 21-46.

Wilkins. D., 1988, Practical Planning: Extending the Classical AI planning paradigm, Morgan Kaufmann, California, USA, ISBN 0-934613-93-X

Wilkins. D., 1990, Can AI planners solve practical problems?, *Computational Intelligence*, vol. 6, no. 4, pp 232-246.

Wilkins. D., 1994, Comparative Analysis of AI Planning Systems, A Report on the AAAI Workshop, AI Magazine, Winter 1994, pp 69-70

Wilkins. D., & Desimone, 1994, Applying an AI Planner to Military Operations Planning, in Zweben. M., & Fox. M. (eds.), Intelligent Scheduling, Morgan-Kaufmann Publishing, San Mateo, California, USA.

Wilkins. D., & Myers. K., 1995, A Common Knowledge Representation for Plan Generation and Reactive Execution. *Journal of Logic and Computation,* vol. 4, part 6.

Williams. B., 1996, Model-Based Autonomous Systems in the New Millennium, In Proceedings of The Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, AAAI Press, ISBN 0-929280-97-0.

Winograd, T., 1971, Procedures as a Representation for Data in a Computer Program for Understanding Natural Language, AI TR-17 (MAC TR-84), MIT-AI-Laboratory, Massachusetts Institute of Technology, Massachusetts, USA.

Winograd. T., and Flores. F., 1986, Understanding Computers and Cognition, Norwood, NJ, Ablex

Winstanley. G., Boardman. J., Kellett. J., 1990, An intelligent planning assistant in the manufacture of flight simulators, In Proceedings of the ACME Research Conference, University of Birmingham, UK, September.

Winstanley. G., and Hoshi. K., 1993, Activity Aggregation in Model-Based AI Planning Systems, *AI in Engineering Design and Manufacture*, Vol. 7. No. 3, pp 209-228, Academic Press.

Wirfs-Brock. R., Wilkerson. B., & Wiener. L., 1990, Design Object-Oriented Software, Prentice Hall, New Jersey, USA, ISBN 0-13-629825-7.

Wolverton. M., & Washington. R., 1996, Segmenting Reactions to improve the Behaviour of a Planning / Reacting Agent., In Proceedings of The Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, AAAI Press, ISBN 0-929280-97-0.

Yang. Q., 1990, Formalising Planning Knowledge for Hierarchical Planning. *Computational Intelligence*, vol. 6, part 1, pp 12-24.

Yang. Q., & Chan. A., 1994, Delaying variable binding commitments in planning, In Proceedings of the Second International conference on Artificial Intelligence Planning Systems, Chicago, USA., pp 182-187

Appendix A

Workbench Algorithms

A. Appendix A - Workbench algorithms

This appendix presents the pseudo code and, where appropriate, the background theory behind in the algorithms implemented within the research workbench described within Chapter 2.

A.1 Management of the partial order within the ADS manager

The algorithms specified in this subsection support the management of plan partial orders within the workbench. The algorithms are implemented, without modification, from the specifications in Fox and Long (1996). Before reproducing the algorithms, Fox and Long's explanation of the theory behind the algorithms is reproduced, and a set of definitions necessary to read the pseudo code provided.

Background theory

"The partial order representation is based on the observation that the subset relation $\subseteq$, is a partial ordering on sets and that any partial order can be embedded in a partial ordering on sets. Given a partial order $p = (S<)$, there is a mapping, $f: S \rightarrow P(S)$, such that for s and t in S, $s < t$ iff $f(s) \subset f(t)$. By careful construction of the sets $f(s)$, for each s in S, the cost of checking whether $s < t$ is kept to the cost of checking a subset relation between comparatively small number of elements ."(Fox and Long 1996)

Definitions of terms

- **Vertex**: each activity in a plan is a vertex
- **Plan Head:** is a special vertex, which is before all other vertices in a plan.
- **Successor List**: each vertex has a list of successors, which includes at least the immediate successors.
- **Predecessor List:** each vertex has a list of predecessors, which includes at least the immediate predecessors.
- **Index:** each vertex has an integer index.
- **Set:** each vertex has an associated set.

The successor list, predecessor list, index, and set of a vertex v will be referred to as succs(v), preds(v), index(v), set(v) respectively.

The Algorithms

```
Add Constraint(t1,t2)
if not(t1 <= t2) then
        if t2 <t1 then
                "error inconsistent constraints"
        else
                if t1 is a new vertex then
                        create a new element for t1
                        add t1 to succs(plan head)
                        if t2 is a new vertex then
                                create a new element for t2
                                add t1 to preds(t2)
                                add t2 to succs(t2)
                                if plan head is newly bifurcating then
                                        identify the existing vertex in succs
                                        (plan head), s
                                        propagate {s} from s
                                        make t1 a key holder with set(t1):= {t1}
                                        set(t2) = set(t1)
                                else if root time is already bifurcating then
                                        make t1 a key holder with set(t1):={t1}
                                        Set(t2):= set(t1)
                                else
                                        set(t1) = set(plan head)
                                        set(t2) = set(root time)
                                end if
                        end if
                        index(t1) = 1
                        index(t2) = 2
                else {t2 is an existing vertex}
                        add t2 to succs(t1)
                        add t1 to preds(t2)
                        index(t1) = 1
                        offset = max(index(t1) - index(t2)+1,0)
                        propagate {t1} from t2 with offset
                        make t1 a key holder with set(t1) = {t1}
                        if plan head is newly bifurcating then
                                identify the existing vertex in succs(plan head), s
                                propagate {s} from S
                        end if
                end if
        else {t1 is an existing vertex}
                if t2 is a new vertex then
                        create a new element for t2
                        add t1 to preds(t2)
                        add t2 to succs(t1)
                        index(t2) = index(t1) + 1
                        if t1 is newly bifurcating then
                                identify existing vertex in succs(t1),s
                                propagate {s} from s
                                make t2 a key holder with set(t2) = set(t2) union
                                 set (t1)
                        else if t1 is already bifurcating then
                                make t2 a key holder with set(t2) = set(t2) union
                                set(t1)
                        else
                                set(t2) = set(t1)
                        end if
                else {t2 is an existing vertex}
                        add t1 to preds(t2)
                        add t2 to succs(t1)
                        offset = max(index(t1)-index(t2) +1,0)
                        if t1 is newly bifurcating then
                                identify the existing vertex in succs(t1),s
```

```
                                        propagate {s} from s
                                        propagate {t2} union set{t1} from t2 with offset
                                else if t1 is already bifurcating then
                                        propagate {t2} union set(t1) from t2 with offset
                        else
                                        propagate set(t1) from t2 with offset
                                end if
                        end if
end if
-- Remove Dead Edges (t1, t2).
-- This line of the algorithm is omitted to allow redundant edges within plans.
-- The line is invoked when exporting the plan to aid readability.
End add constraint.

Propagate set s from vertex v with offset o
if v is not a key holder then
        temp set := set(v)
        make v a key holder with set(v) = a copy of set (v)
end if
for each vertex w, in the partial order
        if v < w then
                if w is a key holder then
                        set(w) = set(w) union S
                else if v was bit a key holder and key(set(w)) = key(temp set) then
                        set(w) = set(v)
                        index(w) = index(w) + o
                end if
        end if
end loop
set(v) = set(v) union S
index(v) = index(v) + o
end propagate.

Remove dead edges (t1, t2)
        for each edge (s, e) in the partial order such that s <= t1 and t2 <= 2 and
        (t1 not equal S or t2 not equal e) loop
                succs(s) = succs(s) / {e}
                preds(e) = preds(e) / {s}
                if s is no longer bifurcating then
                        flatten (e)
                        if t1 element of Succs(s) then
                                flatten(t1)
                        end if
                end if
        end loop
end remove dead edges

flatten(t)
        if t has no bifurcating immediate predecessors then
                for each set, S, associated with a key holder do
                        s = s / {t}
                end loop
        end if
end flatten
```

A.2 Plan state variable manager critic's algorithms

A.2.1 Check co-designation consistency algorithm

The following algorithm checks the co-designation and non co-designation constraints on a variable within the plan state variable manager. It returns true if the constraints are consistent, false otherwise. The algorithm was developed by the author for the workbench, and has not been proved formally correct or complete.

```
Check co-designation Consistency (Variable)
If variable is instantiated then
        If instantiation is consistent with type then
                return true
        else
                return false, type - instantiation inconsistency
        end if
else
        for each constraint on the variable loop

                if the current constraint is set to equal another variable then
                        if the variable current constraint is constrained
                        to equal is itself constraint not to equal the current variable
                then

                                return false - inconsistent constraint
                        else
                                return true
                        end if
                else If current constraint is set to not equal another variable then
                        if the variable current constraint is constrained not to
                        equal is constrained to equal the current variable then

                                return false -- inconsistent constraint
                        else
                                return true
                        end if
                else
                        unknown constraint type error
                end If
        end loop -- for each constraint on the variable
end if -- variable is instantiated
end -- check co-designation consistency
```

A.2.2 Necessarily codesignation algorithm

The following algorithm considers if two statements necessarily co-designate. It takes two arguments of the form "function parameter1.. parameter n". The algorithm is based on the definitions given in (Wilkins 1988, page 72).

```
Necessarily Co-designate (argument1, argument2)
If argument1 is identical to argument2 then
        return true
else
        Pass argument 1 and argument 2 to extract the function names and
        the variables or objects within the statements. E.g. the argument
        on block1, block2 will result in "function = on" and a list of parameters
        [block1, block2]
        If the function of argument1 is not equal to the function of argument2 then
                return false
        else If the number of parameters in argument1 and argument2 is not equal then
                return false
        else
                loop through each parameter, parameterN
                        if argument1.parameterN and argument2.parameterN are
                        objects then
                                if the objects are not equal then
                                        return false
                                else
                                        no problem with this parameter
                                end if
                        else if argument1.parameterN and argument2.parameterN
                        are variables then
                                if they are the same variable or
                                constrained to equal the same thing,
                                with the same constraints then
                                        no problem with this parameter
                                else
                                        return false
                                end if
                        else if argument1.parameterN is a variable and
                argument2.parameterN is an object then
                                if argument1.parameterN is set to equal
                                argument2.parameterN then
                                        no problem with this parameter
                                else
                                        return false
                                end if
                        else if argument1.parameterN is an  object and
                        argument2.parmeterN is a variable then
                                if argument2.parameterN is set to equal
                                argument1.parameterN then
                                        no problem with this parameter
                                else
                                        return false
                                end if
                        end if
                end loop
                if no problems found with any parameter then
                        return true
                else
                        return false
                end if
        end if
end -- necessarily co-designation algorithm
```

A.2.3 Possible co-designation algorithm

The following algorithm considers if two arguments possibly co-designate. The algorithm is based on the definition in (Wilkins 1988 page 72).

```
Possible codesignation (argument1 argument2)

if argument1 equals argument2 then

        return true
else
        Pass argument 1 and argument 2 to extract the function names and
        the variables or objects within the statements. E.g. the argument
        on block1, block2 will result in "function = on" and a list of parameters
        [block1, block2]
        if argument1.function not equal to argument2.function or the
        number of parameters is different then
                return false
        else
                loop through each parameter,
                        if argument1.parameterN = argument2.parameterN then
                                no problem with this parameter
                        else if argument1.parameterN is a variable and
                        argument2.parameterN is an object or a variable then
                                if argument1.parameterN is not constrained to
                                not equal the object in argument2 then
                                        return true if argument1. Parameter has
                                        a constraint added
                                else
                                        return false
                                end if
                        else if argument2.parameterN is a variable and
                        argument1.parameterN is an object or a variable then
                                if argument2.parameterN is not constrained to
                                not equal the object in argument1 then
                                        return true if argumen2.parameterN has
                                        a constraint added
                                end if
                        end if
                end loop
        end if
end --possibly co-designate
```

A.3 Condition and effect manager

The conditions and effect manager functionality is based upon the question and answering system within the Nonlin system as reported in (Tate 1976).

The manager supports two types of question:-

1. Does statement P have value V at node N in a plan network ? The system responds yes, no, or maybe. The maybe response contains a set of constraints which may be added to a plan to make statement P have value V at node N. These may be links or variable binding constraints.
2. What links would have to be added to a network to make P have a value V at node N. Queries of this type are only made as a result of maybe being returned from question 1.

Before introducing the question and answering critic, the following definitions must be made.

- **p-node.** Is a node within a plan which gives statement p a value.
- **pv-node.** Is a node within a plan which gives statement p a value v
- **p!v-node.** is a node within a plan which gives statement p a value other than v.
- **critical node.** A critical node for (P,N) is a node which gives a value to statement p for the last time before node N. i.e. it could be made the nearest predecessor of N which gives a value to p in some legal linearization of the partially-ordered network of nodes. The critical nodes for (P,N) are
    1. the last p-node on each incoming branch. ignore p-nodes which are also predecessors of any other critical nodes.
    2. all p-nodes which are in parallel with n.

```
QA(P, V, N)
        Identify the following lists of nodes
                vl. Critical pv-nodes which are predecessors of n
                v!l. critical p!v-nodes which are predecessors of n
                par-vl critical pv-nodes which are in parallel with n.
                par-v!l critical p!v-nodes which are in parallel with n.

        If at least one member of vl and no par-v!l nodes then

                return YES

        else if at least one member of par-v!l and no vl nodes.

                return NO

        else if it is possible to create Vl and remove par-v! nodes with
        variable bindings then

                return maybe and constraints,

        else consider adding new links

                return Maybe and result of calling the linking
                call linking algorithm and return results

        end if
end -- QA
```

The algorithm differentiates between possible and necessary co-designation when constructing the four lists at the top of the algorithm. If a condition possibly co-designates with the statement P, and is critical, it is added to the critical lists together with the constraints to make it necessarily co-designate.

A special case of this algorithm is invoked when handling plans with condition types. Only conditions of type supervised, achieve, and only_use_for_query are checked.

A.4 HTN Engine Algorithm

A.4.1 Controller Algorithm

```
Read problem definition and task formalism schemas
2.    Append non-primitive tasks from task definition to the task queue
3.    Append achieve conditions from task definition to the task queue
4.    While the task queue is not empty loop
5.            Select a task for processing (either at random, FIFO, or ask user)
6.            if selected task is a non-primitive task then
7.                    -- task reduction procedure
8.                    Ask schema library for methods to achieve selected task
9.                    remove methods whose only_use_if conditions do not
10.                   hold
11.                   if no methods available then
12.                           return to select a task for processing
13.                   end if -- no methods available
14.                   select a method from the set remaining (either at random
                      or ask user)
15.                   expand the selected task with the selected method
16.           else -- it must be an achieve task
17.                   if it is possible to add links to make condition true then
18.                           add links
19.                   else
20.                           ask schema library for schema to make condition
                              true
21.                           if not methods available then
22.                                   return to select a task for processing
23.                           end if -- no methods available
24.                           implement schema immediately before task
25.                           which requires condition
26.                   end if -- it is possible to add links...
27.           end if -- selected task is a non-primitive task
28.           call plan state variable critic
29.           loop until critic replies yes to all variables or user quits
30.           loop
31.                   allow user to correct problems
32.           end loop - PSVM critic
33.           call conditions and effects critique
34.           loop until critic replies yes to all conditions or user quits
35.           loop
36.                   allow user to correct problems
37.           end loop - conditions and effects critique
38.   end loop -- while task the task queue is not empty
39.   Call PSVC to check unsupervised conditions.
```

A.4.2 Expand node algorithm

```
expand node (node, schema name)

        record the current predecessors and successors of node
        remove node from the ADS
        insert all the nodes from schema name  into the ADS
        Insert all the non-primitive new nodes into the task queue.
        Add ordering constraints from scheme name
        Add conditions from schema name
        Add effects from schema name

        If expansion has no single start node then
                create a dummy node and make dummy the start node
        end if

        if new expansion has no single terminal node then
                create a dummy node and make dummy the terminal
                node
        end if

        Replace all entries in the ADS which refer to node with the
        Expansions start node and terminal node.
        Attach conditions from node to expansions start node
        Attach effects from node to expansions terminal node
```

The expand node algorithm introduces dummy actions when an expansion does not have a single start or finish node. This mechanism allow the conditions and effects of the node being replaced to be attached to a definite point in the refined task network.

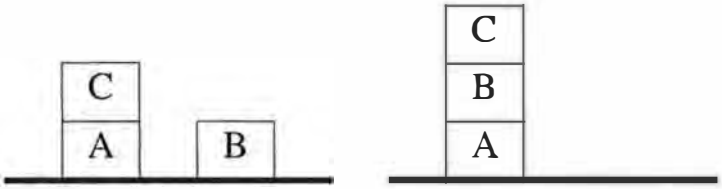
Blank

Appendix B

Workbench Testing

B. The Sussman Anomaly

The Sussman Anomaly is a classic benchmark problem in AI planning literature, which demonstrates the issues encountered when addressing conjunctive goals. Before expanding this point, the initial and goal states of the problem are introduced..



Initial State Goal State

The anomaly contains two conjunctive goals: {on (BlockB BlockA) ∧ on (BlockC BlockB) }. If a planner addresses the on(BlockC BlockB) goal first, the second goal is no longer immediately obtainable, as BlockB is obstructed by BlockC, and therefore cannot be moved onto BlockA. The planner must undo its solution to the goal on(BlockC BlockB) before BlockB can be moved.

The anomaly was selected as a test case for the workbench as it demonstrates the precondition achievement behaviour, and plan critic functions of the workbench. The remainder of this section steps through the workbench's operation whilst solving this problem.

B.1 Experiment set up

The Sussman Anomaly was specified using the following task definition from the O-Plan Task Formalism manual.

```
task stack_BlockA_BlockB_BlockC;
        nodes           1       start,
                        2       finish;
        orderings 1->2;
        conditions      achieve {on BlockA BlockB} at 2,
                                {on BlockB BlockC} at 2;
        effects         {on BlockC BlockA} at 1,
                        {on BlockA TheTable} at 1,
                        {on BlockB TheTable} at 1,
                        {cleartop BlockC} at 1,
                        {cleartop BlockB} at 1;
end_task;
```

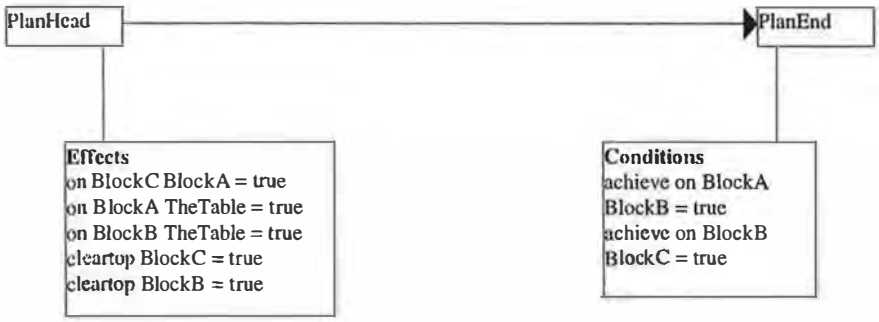
The domain's action knowledge was supplied to the workbench through the following task formalism file.

```
always          {clear top TheTable};

types           objects = {BlockA BlockB BlockC TheTable},
                movable_objects = {BlockA BlockB BlockC};

schema puton;
      vars      ?x = ?{type movable_objects},
                ?y = ?{type objects},
                ?z = ?{type objects};

      var_relations
                ?x /= ?y, ?y /= ?z, ?x /= ?z;

      expands {puton ?x ?y};

      only_use_for_effects
                {on ?x ?y}          = true,
                {cleartop ?y}       = false,
                {on ?x ?z}          = false,
                {cleartop ?z}       = true;

      conditions
                only_use_for_query          {on ?x ?z}
                achieve                     {cleartop ?y}
                achieve                     {cleartop ?x}
end_schema;
```

B.2 Experiment Execution

Initialisation

The domain representation was passed to the schema library, and the HTN engine initiated with the task *stack_BlockA_BlockB_BlockC*. In response to this initialisation, the workbench generated the following plan.



Step 1

The HTN engine called the QA algorithm to determine if either of the conditions on the activity PlanEnd was satisfied. The algorithm replied NO in both cases. The HTN engine added *on BlockA BlockB* and *on BlockB BlockC* to the task queue as conditions to be achieved.

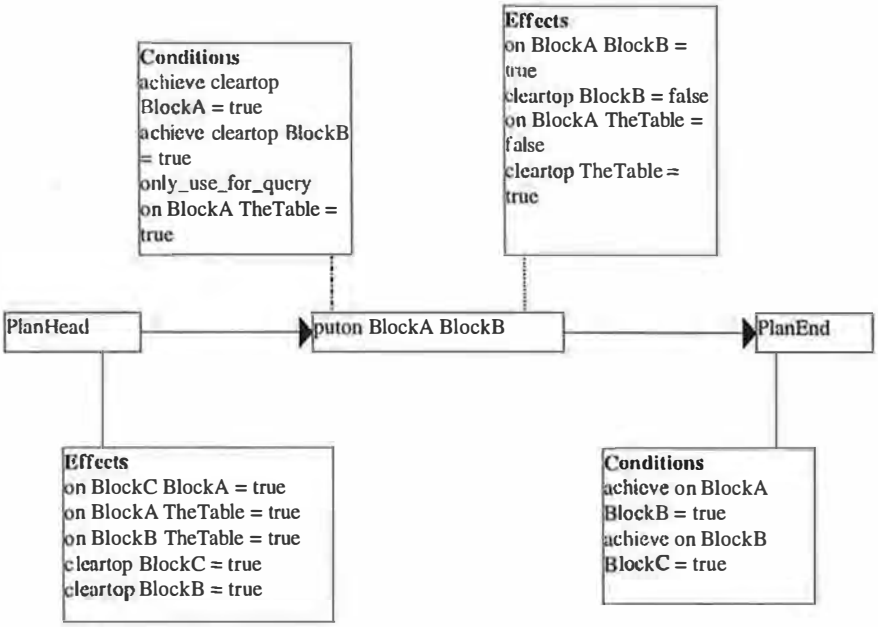
Step 2

With no outstanding non-primitive tasks for refinement, the HTN engine selected the condition *achieve on BlockA BlockB* for resolution. The *achieve* condition type permits the planner to include new tasks into a plan to satisfy the condition it prefixes. The workbench selected the *On BlockA BlockB* condition for attainment. The selection order is a result of the order of the conditions in the initial task description; the workbench employees a first in first out data structure for storing outstanding conditions.

The HTN engine issues a request to the schema library for tasks which will make the effect *on BlockA BlockB true*. The library returned the schema puton as it contains the effect *on ?x ?y = true*. Pattern matching this effect against the condition resulted in the following variable bindings.

?x = BlockA from pattern match
?y = BlockB from pattern match
?z= TheTable from the only_use_for_query condition. TheTable is the only variable binding option to make this condition true.

The task added to establish an achieve condition are automatically linked immediately before the requiring activity. The workbench produced the following plan:



The HTN engine the applied the critics to the plan, with the following results.

Condition	Must hold at	Status	Constraints
On BlockB BlockC = true	plan end	***no***	Close world assumption
on BlockA BlockB = true	plan end	yes	from puton BlockA BlockB
on BlockA TheTable = true	puton BlockA BlockB	yes	from plan head
cleartop BlockA = true	puton BlockA BlockB	***no***	closed world assumption
cleartop BlockB = true	puton BlockA BlockB	yes	from plan head

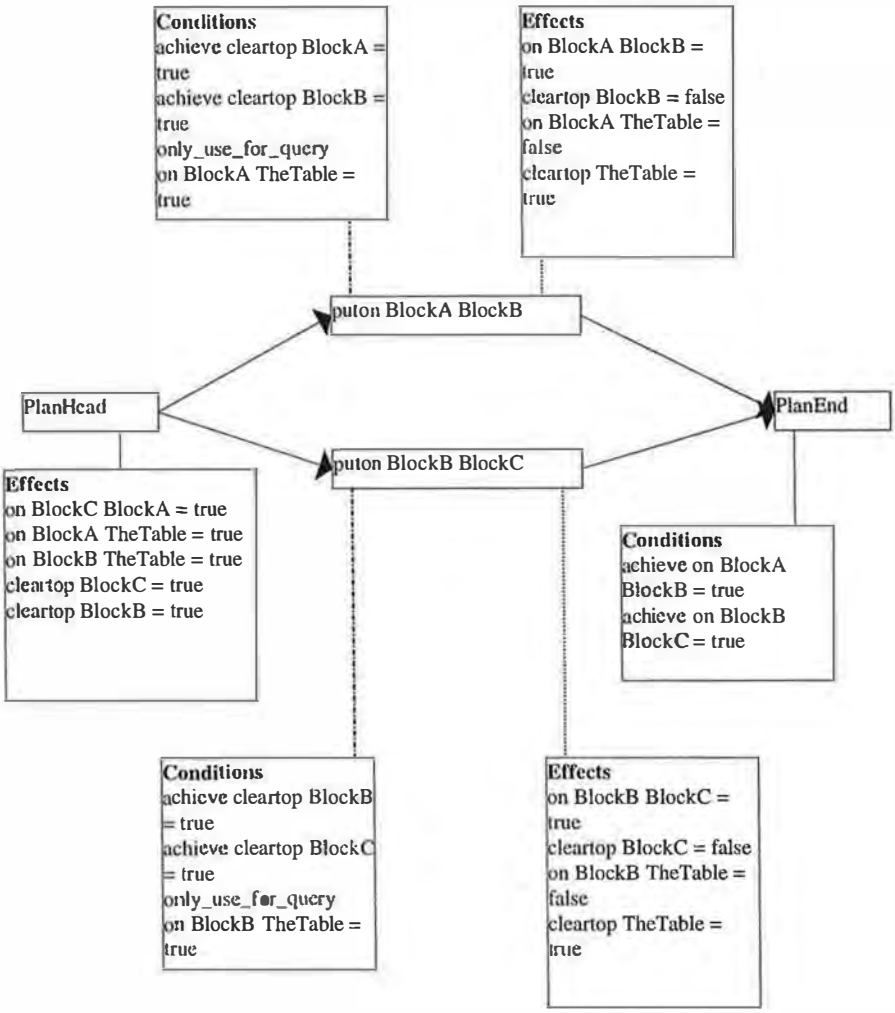
The resulting plan contains two flaws (marked with a status no in the table): the second condition on the goal state of the plan, and a precondition of the new task added to the plan.

Step 3

The achieve condition *on BlockB BlockC* is the next flaw in the queue, and was therefore selected for processing next. The HTN engine issues a request to the schema library for tasks which will make the effect *on BlockB BlockC true.* The library returned the schema *puton* as it contains the *effect on ?x ?y = true.* Pattern matching this effect against the condition resulted in the following variable bindings.

?x = BlockB
?y = BlockC
?z = TheTable from the only_use_for_query condition. TheTable is the only variable binding option to make this condition true.

The task added to establish an achieve condition are linked immediately before the requiring activity. The workbench produced the following plan:



The HTN engine then applied the critics to the plan with the following results.

Statement	Must Hold At	Status	Constraints
on BlockB BlockC = true	at plan end	yes	from node puton BlockB BlockC
on BlockA BlockB = true	at plan end	yes	from node puton BlockA BlokcB
on BlockA TheTable = true	at puton BlockA BlockB	yes	from plan head
cleartop BlockA = true	at puton BlockA BlockB	***no***	Closed world asumption
cleartop BlockB = true	at puton BlockA BlockB	yes	from plan head
on BlockB TheTable = true	at puton BlockB BlockC	yes	from plan head
cleartop BlockB = true	at puton BlockB BlockC	***no***	negation in parellel
cleartop BlockC = true	at puton BlockB BlockC	yes	achived by plan head

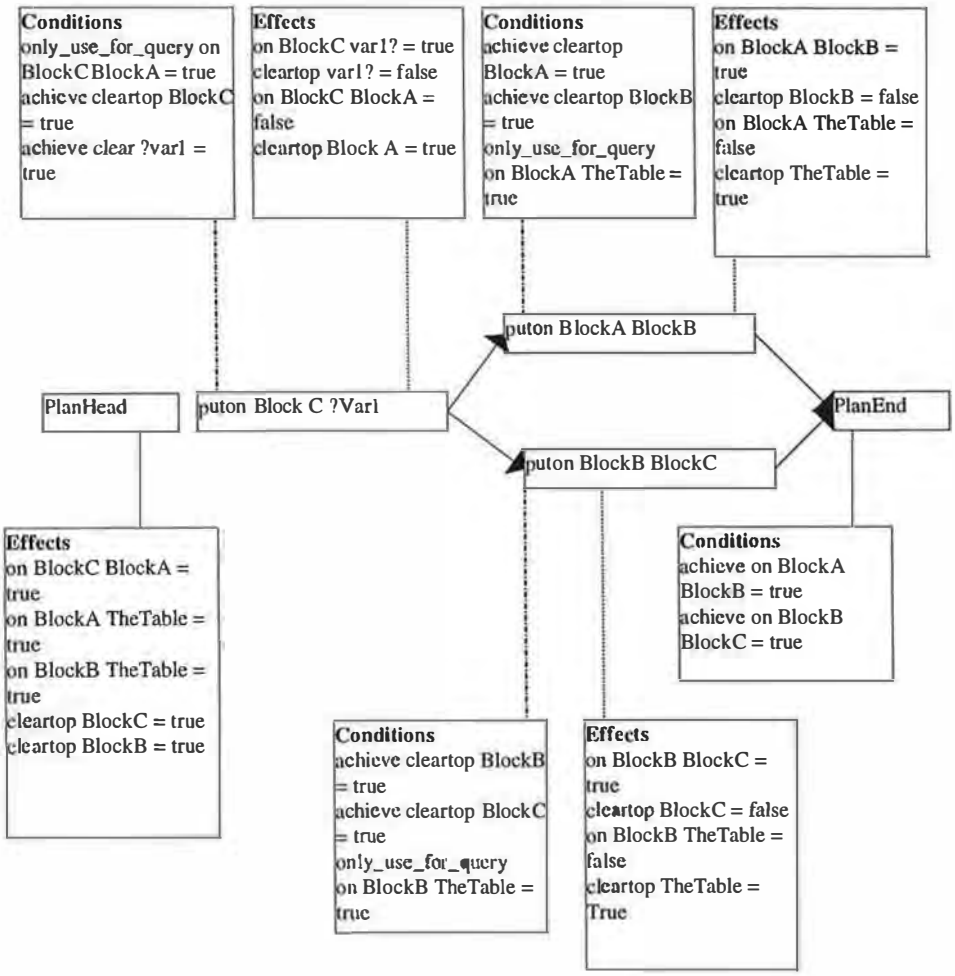
Both conditions on the plan's final state now hold. Two achieve conditions from the tasks introduced to achieve these conditions now do not hold.

Step 4

The achieve condition *Clear BlockA = true* at the task *puton BlockA BlockB* was selected from the queue for processing. The HTN engine issued a request to the schema library for tasks which asserts the effect *cleartop BlockA =true*. The library returned the schema puton as it contains the effect *cleartop ?z= true*. Pattern matching this effect against the condition resulted in the following variable bindings.

x = BlockC, from the only_use_for_query condition.
y = ? unknown
z = BlockA

The HTN engine applied the expansion to the plan, immediately before the task requiring the condition. The resultant plan was as follows:



The HTN engine applied the plan critics to this plan with the following result.

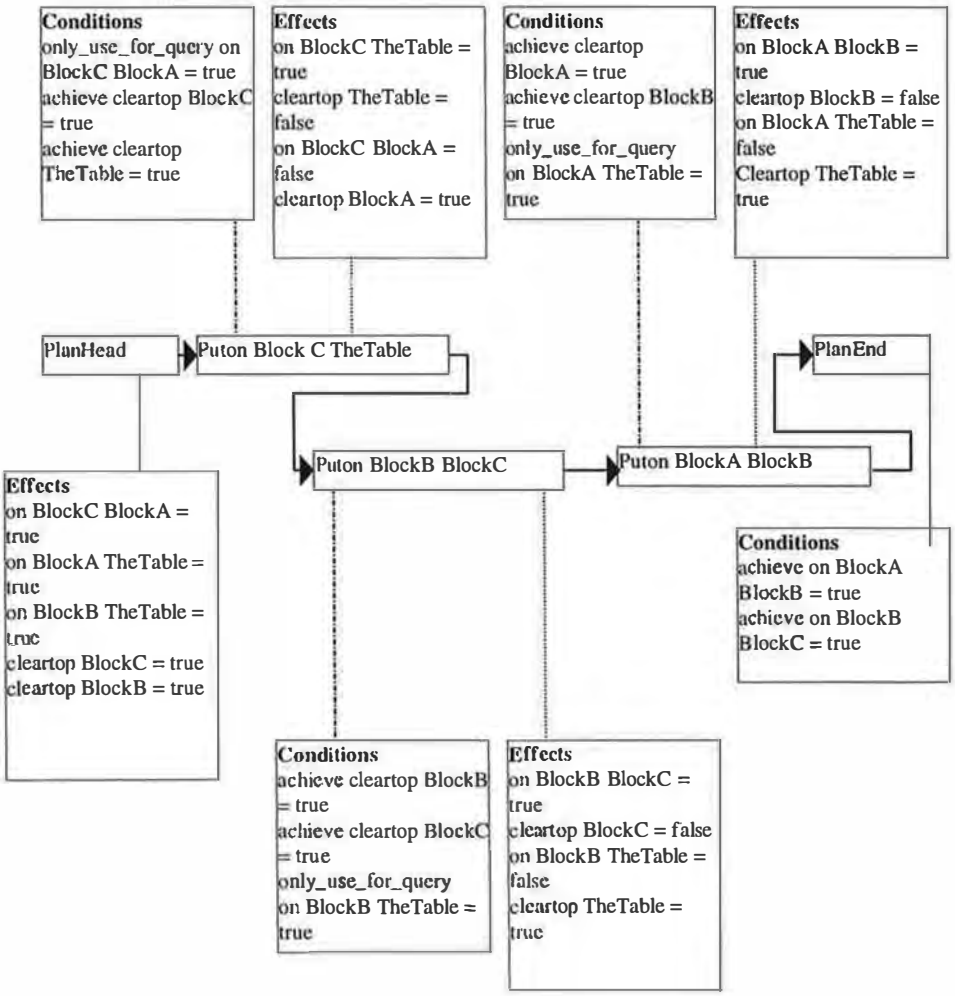
Statement	Must hold at	status	constraints
on BlockB BlockC = true	at plan end	yes	from node puton BlockB BlockC
on blockA BlockB = true	at plan end	yes	from node puton BlockA BlockB
on BlockA TheTable = true	at puton BlockA BlockB	yes	from plan head
cleartop BlockA = true	at puton BlockA BlockB	yes	from puton BlockC ?varl
cleartop BlockB = true	at puton BlockA BlockB	yes	from plan head
on BlockB TheTable = true	at puton BlockB BlockC	yes	from plan head
cleartop BlockB = true	at puton BlockB BlockC	***no***	negation in parellel
cleartop BlockC = true	at puton BlockB BlockC	yes	achived by plan head
on BlockC BlockA = true	at puton BlockC ?varl	yes	from plan head
cleartop BlockC = true	at puton BlockC ?varl	yes	from plan head
cleartop ?varl = true	at puton BlockC ?varl	***maybe***	if ?varl = TheTable or BlockB

The condition *cleartop ?var1* is identified by the condition and effect critique as attainable, with the addition of a variable binding constraint. The HTN engine selected the *TheTable* instantiation as this would bind the condition to an always fact. I.e. it cannot be made false by any effect within the plan.

With the binding made, the condition and effect manager was called to process the plan. Only one condition remained open, the parallel interaction between the *cleartop BlockB* condition at *puton BlockB BlockC* and *Puton BlockA BlockB.*

Step 5

The user was invited to resolve the parallel interaction. This could be achieved by linearising *puton BlockB BlockC* and *puton BlockA BlockB*. Placing *puton BlockA BlockB* before *puton BlockB BlockC* introduced a new interaction, the *cleartop BlockB* condition of *puton BlockA BlockB* was deleted. The alternative ordering introduced no interactions. The completed plan is depicted below.



The final result of the question and answering process below indicates that all conditions are now established.

Statement	Must hold at	status	constraints
on BockB BlockC = true	at plan end	yes	from node puton BlockB BlockC
on blockA BlockB = true	at plan end	yes	from node puton BlockA BlokcB
on BlockA TheTable = true	at puton BlockA BlockB	yes	from plan head
cleartop BlockA = true	at puton BlockA BlockB	yes	from puton BlockC TheTable
cleartop BlockB = true	at puton BlockA BlockB	yes	from plan head
on BlockB TheTable = true	at puton BlockB BlockC	yes	from plan head
cleartop BlockB = true	at puton BlockB BlockC	yes	
cleartop BlockC = true	at puton BlockB BlockC	yes	achived by plan head
on BlockC BlockA = true	at puton BlockC TheTable	yes	from plan head
cleartop BlockC = true	at puton BlockC TheTable	yes	from plan head
cleartop TheTable = true	at puton BlockC TheTable	yes	

With all conditions established, planning is complete.

B.3 Tate's house building domain

The house building domain originates in (Tate 1976) and is designed to demonstrate the task refinement function of Nonlin and recently the O-Plan system. The domain was selected to test the workbench's task refinement functionality.

Two sets of tests were undertaken. The first set without the plan critics operating, and the second set with the critics operating. The first experiment verified the pure task refinement function of the workbench. i.e. it can correctly replace a task with its refinement. The second test set verified the complete workbench architecture.

This section reports the first experiment set.

B.4 Experiment set up

The build house task was specified using the following task definition. The definition is taken directly from the O-Plan test specifications on the World Wide Web (url http://www.aiai.ed.ac.uk/cgi-bin/oplan/web-demo/show-tf/house-1.tf [accessed 11 Apr 1997]).

```
task build_house;
        nodes           1       start,
                        2       finish,
                        3       action {build house};
        orderings       1 ---> 3, 3 ---> 2;
end_task;
```

The definitions of the operators available in the domain were provided using the following domain description taken directly from the O-Plan World Wide Web site (url http://www.aiai.ed.ac.uk/cgi-bin/oplan/web-demo/show-tf/house-1.tf [accessed 11 Apr 1997]).

```
;;; House Building Domain - with no schema choice
;;;
;;; BAT   5-Dec-76: Nonlin TF.
;;; KWC   9-Sep-85: Converted to O-Plan1 TF.
;;; BD   12-May-92: Converted to O-Plan2 TF.
;;;
schema build;
 expands {build house};        ;;; this expands the top level action
 nodes    1 action {excavate and pour footers    }, ;;; some are primitive
          2 action {pour concrete foundations    },
          3 action {erect frame and roof         },
          4 action {lay brickwork                },
          5 action {finish roofing and flashing  },
          6 action {fasten gutters and downspouts},
          7 action {finish grading               },
          8 action {pour walks and landscape     },
          9 action {install services             }, ;;; some are not.
         10 action {decorate                     };
 orderings 1 ---> 2, 2 ---> 3, 3 ---> 4, 4 ---> 5,
          5 ---> 6, 6 ---> 7, 7 ---> 8;
 ;;; actions 9 & 10 are not ordered wrt other actions - they are in parallel
 conditions   supervised {footers poured        } at 2 from [1],
              supervised {foundations laid      } at 3 from [2],
              supervised {frame and roof erected} at 4 from [3],
              supervised {brickwork done        } at 5 from [4],
              supervised {roofing finished      } at 6 from [5],
              supervised {gutters etc fastened  } at 7 from [6],
            unsupervised {storm drains laid     } at 7,
              supervised {grading done          } at 8 from [7];
 ;;; note the unsupervised condition - its satisfaction is outwith
 ;;; the control of this schema but must still be satisfied
end_schema;
schema service_1;
 expands {install services};
 only_use_for_effects {installed services 1};
 nodes    1 action {install drains          },
          2 action {lay storm drains        },
          3 action {install rough plumbing  },
          4 action {install finished plumbing},
          5 action {install rough wiring    },
          6 action {finish electrical work  },
          7 action {install kitchen equipment},
          8 action {install air conditioning };
 orderings 1 ---> 3, 3 ---> 4, 5 ---> 6, 3 ---> 7, 5 ---> 7;
 conditions   supervised {drains installed        } at 3 from [1],
              supervised {rough plumbing installed} at 4 from [3],
              supervised {rough wiring installed  } at 6 from [5],
              supervised {rough plumbing installed} at 7 from [3],
              supervised {rough wiring installed  } at 7 from [5],
            unsupervised {foundations laid        } at 1,
            unsupervised {foundations laid        } at 2,
            unsupervised {frame and roof erected  } at 5,
            unsupervised {frame and roof erected  } at 8,
            unsupervised {basement floor laid     } at 8,
            unsupervised {flooring finished       } at 4,
            unsupervised {flooring finished       } at 7,
            unsupervised {painted                 } at 6;
   ;;; As in the real world this sub-contractor relies heavily on others
   ;;; to prepare things beforehand - see the unsupervised conditions.
end_schema;

schema decor;
 expands {decorate};
 nodes    1 action {fasten plaster and plaster board}.
```

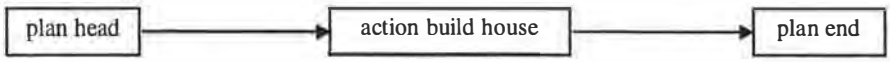
```
        2 action {pour basement floor          },
        3 action {lay finished flooring        },
        4 action {finish carpentry             },
        5 action {sand and varnish floors      },
        6 action {paint                        };
 orderings  2 ---> 3,  3 ---> 4,  4 ---> 5,  1 ---> 3,  6 ---> 5;
 conditions unsupervised {rough plumbing installed   } at 1,
          unsupervised {rough wiring installed    } at 1,
          unsupervised {air conditioning installed } at 1,
          unsupervised {drains installed          } at 2,
          unsupervised {plumbing finished         } at 6,
          unsupervised {kitchen equipment installed} at 6,
           supervised {plastering finished        } at 3 from [1],
           supervised {basement floor laid        } at 3 from [2],
           supervised {flooring finished          } at 4 from [3],
           supervised {carpentry finished         } at 5 from [4],
           supervised {painted                    } at 5 from [6];
end_schema;
;;; Now for completeness a list of primitive actions. Primitives are
;;; defined as having no nodes list and must have an expands pattern.
schema excavate;
 expands {excavate and pour footers};
 only_use_for_effects {footers poured} = true;
end_schema;
schema pour_concrete;
 expands {pour concrete foundations};
 only_use_for_effects {foundations laid} = true;
end_schema;
schema erect_frame;
 expands {erect frame and roof};
 only_use_for_effects {frame and roof erected} = true;
end_schema;
schema brickwork;
 expands {lay brickwork};
 only_use_for_effects {brickwork done} = true;
end_schema;
schema finish_roofing;
 expands {finish roofing and flashing};
 only_use_for_effects {roofing finished} = true;
end_schema;
schema fasten_gutters;
 expands {fasten gutters and downspouts};
 only_use_for_effects {gutters etc fastened} = true;
end_schema;
schema finish_grading;
 expands {finish grading};
 only_use_for_effects {grading done} = true;
end_schema;
schema pour_walks;
 expands {pour walks and landscape};
 only_use_for_effects {landscaping done} = true;
end_schema;
schema install_drains;
 expands {install drains};
 only_use_for_effects {drains installed} = true;
end_schema;
```

```
schema lay_storm;
  expands {lay storm drains};
  only_use_for_effects {storm drains laid} = true;
end_schema;
schema rough_plumbing;
  expands {install rough plumbing};
  only_use_for_effects {rough plumbing installed} = true;
end_schema;
schema install_finished;
  expands {install finished plumbing};
  only_use_for_effects {plumbing finished} = true;
end_schema;
schema rough_wiring;
  expands {install rough wiring};
  only_use_for_effects {rough wiring installed} = true;
end_schema;
schema finish_electrical;
  expands {finish electrical work};
  only_use_for_effects {electrical work finished} = true;
end_schema;
schema install_kitchen;
  expands {install kitchen equipment};
  only_use_for_effects {kitchen equipment installed} = true;
end_schema;
schema install_air;
  expands {install air conditioning};
  only_use_for_effects {air conditioning installed} = true;
end_schema;
schema fasten_plaster;
  expands {fasten plaster and plaster board};
  only_use_for_effects {plastering finished } = true;
end_schema;
schema pour_basement;
  expands {pour basement floor};
  only_use_for_effects {basement floor laid } = true;
end_schema;
schema lay_flooring;
  expands {lay finished flooring};
  only_use_for_effects {flooring finished} = true;
end_schema;
schema finish_garden;
  expands {finish garden};
  only_use_for_effects {garden finished};
end_schema;
schema finish_carpentry;
  expands {finish carpentry};
  only_use_for_effects {carpentry finished} = true;
end_schema;
schema sand;
  expands {sand and varnish floors};
  only_use_for_effects {floors finished} = true;
end_schema;
schema paint;
  expands {paint};
  only_use_for_effects {painted} = true;
end_schema;
```

B.5 Experiment Execution

Initialisation

The domain representation was passed to the schema library, and the HTN engine initiated with the task "build_house". In response to this initialisation, the workbench generated the following plan.



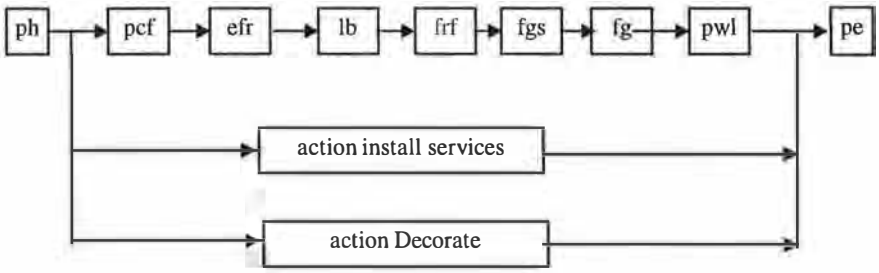
The planners task queue was set as follows:

Task Queue
action build house

Step 1

The planner has one outstanding task: *build house*. The schema library was requested to provide candidates schemas for achieving this task. One schema, *build*, was returned. As plan critics were not active, the schema's *only_use_if* conditions were not checked.

The planner replaced the build house task in the initial plan with the partial plan described in the schema build. The resultant plan is depicted below.



The resultant task queue was as follows. The shaded box indicates the task removed during the expansion. The abbreviations in the plan above are attached to the full names of the tasks.

Task Queue
action build house
action pour concrete foundations (pcf)
action erect frame and roof (efr)
action lay brickwork (lb)
action finish roofing and flashing (frf)
action fasten gutters and downspouts (fgs)
action finish grading (fg)
action pour walks and landscape (pwl)
action install services
action decorate

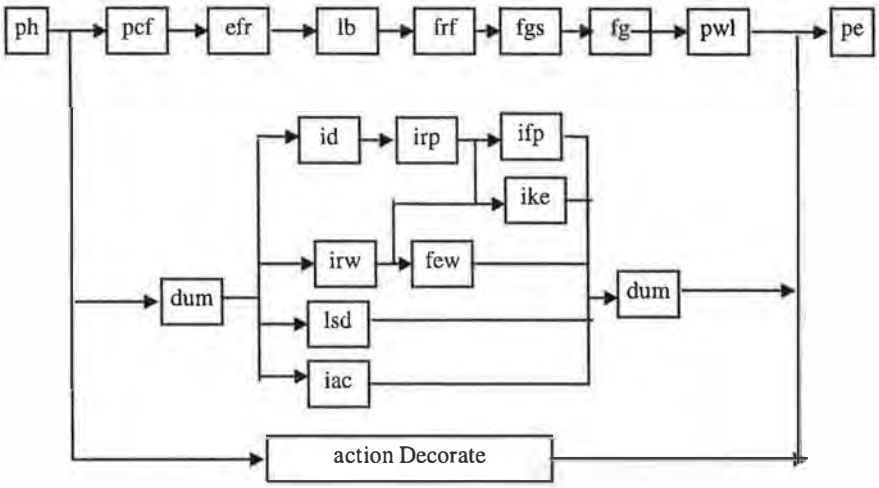
Step 2

The first seven tasks on the task queue refined immediately to primitive tasks. The task queue after processing was as follows:

Task Queue
action pour concrete foundations (pcf)
action erect frame and roof (efr)
action lay brickwork (lb)
action finish roofing and flashing (frf)
action fasten gutters and downspouts (fgs)
action finish grading (fg)
action pour walks and landscape (pwl)
action install services
action decorate

Step 3

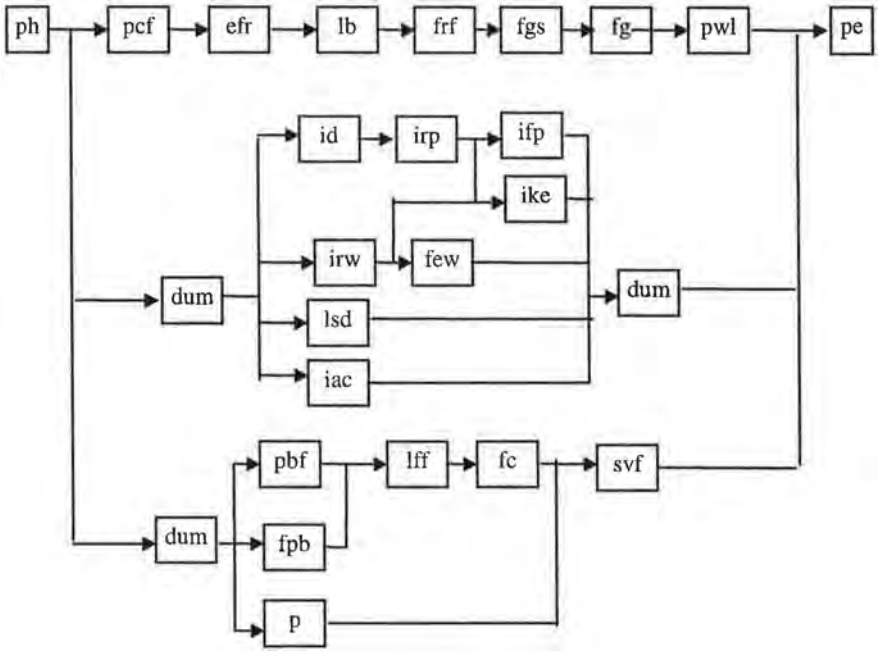
The schema library returned one task for refining "install services". The install services task was replaced with this method, resulting in the following plan.



Task Queue
action install services
action decorate
action install drains (id)
action install rough plumbing (irp)
action install finished plumbing (ifp)
action install rough wiring (irw)
action finish electrical work (few)
action install kitchen (ike)
action install air conditioning (iac)

Step 4

The next task is decorate and this is what we get.



The expansion resulted in the following task queue.

Task Queue
action decorate
action install drains (id)
action install rough plumbing (irp)
action install finished plumbing (ifp)
action install rough wiring (irw)
action finish electrical work (few)
action install kitchen (ike)
action install air conditioning (iac)
action fasten plaster and plaster board (fpb)
action pour basement floor (pbf)
action lay finish flooring (lff)
action finish carpentry (fc)
action sand and varnish floors (svf)
action paint (p)

Step 5

All the tasks within the task network at the end of *step 4* refine to primitive tasks. Planning may therefore be considered complete.

Blank

Appendix C

Task Formalism domain specifications

C.1 Full house building domain specification

```
;;; House Building Domain - larger house building example
;;;
;;; BAT  5-Dec-76: Nonlin TF.
;;; KWC  9-Sep-85: Converted to O-Plan1 TF.
;;; BD  12-May-92: Converted to O-Plan2 TF.
;;; BAT  30-Nov-92: remove multiple landscape schema names
;;; BAT  30-Jun-93: task schema only_use_for_effect removed

task build_large_house;
 nodes 1 start,
       2 finish,
       3 action {build house};
 orderings 1 ---> 3, 3 ---> 2;
end_task;

schema build;
 expands {build house};
 nodes    1 action {obtain building permit},
          2 action {lay foundations},
          3 action {build walls and roof},
          4 action {joinery},
          5 action {decorate and fit},
          6 action {install services},
          7 action {landscape},
          8 action {close out house};
 orderings 1 ---> 2,  2 ---> 3,  2 ---> 4,  2 ---> 5,
          2 ---> 6,  2 ---> 7,  3 ---> 8;
 conditions unsupervised {wooden frame and roof erected} at 5;
end_schema;

schema lay_foundations;
 expands {lay foundations};
 nodes    1 action {clear lot and grade for slab},
          2 action {place concrete forms reinforcement rods and sewer lines},
          3 action {pour slab};
 orderings 1 ---> 2,  2 ---> 3;
 effects {foundations laid};
end_schema;

schema build_walls_and_roof;
 expands {build walls and roof};
 nodes    1 action {erect wooden frame including roof},
          2 action {fasten exterior sheathing},
          3 action {insulate outside walls},
          4 action {sheetrock and plaster inside walls},
          5 action {place insulation in attic},
          6 action {attach gutters and downspouts},
          7 action {shingle roof},
          8 action {lay brickwork exterior walls plus inside fireplace};
 orderings 1 ---> 2, 2 ---> 3, 3 ---> 4, 4 ---> 5, 5 ---> 6, 1 ---> 7,
                7 ---> 3,  2 ---> 8,  8 ---> 5;
 conditions unsupervised {foundations laid} at 1,
          unsupervised {rough plumbing installed} at 3,
          unsupervised {rough wiring installed} at 3,
          unsupervised {exterior trim complete} at 8;
end_schema;
```

```
schema joinery;
 expands {joinery};
  nodes    1 action {do rough carpentry including window and door frames},
           2 action {do finish carpentry cabinets trim mouldings panelling},
           3 action {sand stain and varnish wood panelling and cabinets},
           4 action {lay formica counter surfaces in kitchen};
  orderings 1 ---> 2, 2 ---> 3, 3 ---> 4;
  conditions unsupervised {exterior sheathing fastened} at 1,
           unsupervised {plastering done} at 2;
end_schema;

schema landscape;
  expands {landscape};
  nodes    1 action {grade lay forms for walks and driveways},
           2 action {pour walks and driveway},
           3 action {finish grading},
           4 action {landscape_yard};
  orderings 1 ---> 2, 2 ---> 3, 3 ---> 4;
  conditions unsupervised {brickwork laid} at 1,
           unsupervised {interior and exterior cleaned} at 3;
end_schema;

schema install_services;
  expands {install services};
  nodes    1 action {electrical services},
           2 action {plumbing services},
           3 action {install kitchen appliances},
           4 action {heating and air conditioning};
  orderings 2 ---> 3;
  conditions unsupervised {interior painted} at 3;
end_schema;

schema electrical_services;
  expands {electrical services};
  nodes    1 action {install rough wiring},
           2 action {install electrical outlets switches lighting fixtures},
           3 action {final hookup of electrical system};
  orderings 1 ---> 2, 2 ---> 3;
  conditions unsupervised {wooden frame and roof erected} at 1,
           unsupervised {selected surfaces wallpapered} at 2,
           unsupervised {kitchen appliances installed} at 2,
           unsupervised {hot water heater installed} at 2,
           unsupervised {furnace and air conditioner installed} at 2;
end_schema;

schema plumbing_services;
  expands {plumbing services};
  nodes    1 action {install rough plumbing},
           2 action {install tubs and shower basins},
           3 action {install remaining plumbing fixtures},
           4 action {install hot water heater};
  orderings 1 ---> 2, 2 ---> 3,  1 ---> 4;
  conditions unsupervised {wooden frame and roof erected} at 1,
           unsupervised {plastering done} at 2,
           unsupervised {rough carpentry done} at 2,
           unsupervised {bathroom tiles laid} at 3,
           unsupervised {formica surfaces done} at 3;
end_schema;
```

```
schema heating_and_ac;
  expands {heating and air conditioning};
  nodes    1 action {install heating and cooling ducts},
           2 action {install furnace and air conditioner};
  orderings 1 ---> 2;
  conditions unsupervised {wooden frame and roof erected} at 1,
             unsupervised {rough wiring installed} at 2;
  effects {heating and air conditioning installed};
end_schema;

schema decorate;
  expands {decorate and fit};
  nodes    1 action {lay bathroom tiles},
           2 action {sand and paint interior walls and trim},
           3 action {lay flooring wood and vinyl},
           4 action {wallpaper selected surfaces},
           5 action {complete exterior trim},
           6 action {paint exterior trim},
           7 action {clean up interior and exterior including yard},
           8 action {lay carpeting},
           9 action {attach cabinet fixtures};
  orderings 5 ---> 6, 6 ---> 7, 7 ---> 8, 2 ---> 3, 3 ---> 7, 2 ---> 4,
                2 ---> 9;
  conditions unsupervised {tubs and shower basins installed} at 1,
             unsupervised {plumbing finished} at 2,
             unsupervised {exterior sheathing fastened} at 5,
             unsupervised {gutters and downspouts attached} at 6,
             unsupervised {heating and air conditioning installed} at 7;
end_schema;

schema obtain_permit;
  expands {obtain building permit};
  only_use_for_effects {got permit} = true;
end_schema;

schema clear_and_grade;
  expands {clear lot and grade for slab};
  only_use_for_effects {lot cleared} = true;
end_schema;

schema place_forms;
  expands {place concrete forms reinforcement rods and sewer lines};
end_schema;

schema pour_slab;
  expands {pour slab};
end_schema;

schema erect_wooden_frame;
  expands {erect wooden frame including roof};
  only_use_for_effects {wooden frame and roof erected} = true;
end_schema;

schema fasten_exterior_sheathing;
  expands {fasten exterior sheathing};
  only_use_for_effects {exterior sheathing fastened} = true;
end_schema;

schema install_rough_plumbing;
  expands {install rough plumbing};
  only_use_for_effects {rough plumbing installed} = true;
end_schema;
```

```
schema install_rough_wiring;
  expands {install rough wiring};
  only_use_for_effects {rough wiring installed} = true;
end_schema;

schema insulate_outside;
  expands {insulate outside walls};
end_schema;

schema plaster_inside;
  expands {sheetrock and plaster inside walls};
  only_use_for_effects {plastering done} = true;
end_schema;

schema rough_carpentry;
  expands {do rough carpentry including window and door frames};
  only_use_for_effects {rough carpentry done} = true;
end_schema;

schema finish_carpentry;
  expands {do finish carpentry cabinets trim mouldings panelling};
end_schema;

schema sand_and_varnish;
  expands {sand stain and varnish wood panelling and cabinets};
end_schema;

schema do_kitchen_surfaces;
  expands {lay formica counter surfaces in kitchen};
  only_use_for_effects {formica surfaces done} = true;
end_schema;

schema install_tubs;
  expands {install tubs and shower basins};
  only_use_for_effects {tubs and shower basins installed} = true;
end_schema;

schema do_tiles;
  expands {lay bathroom tiles};
  only_use_for_effects {bathroom tiles laid} = true;
end_schema;

schema install_remaining_plumbing;
  expands {install remaining plumbing fixtures};
  only_use_for_effects {plumbing finished} = true;
end_schema;

schema sand_and_paint;
  expands {sand and paint interior walls and trim};
  only_use_for_effects {interior painted} = true;
end_schema;

schema lay_flooring;
  expands {lay flooring wood and vinyl};
end_schema;

schema do_wallpaper;
  expands {wallpaper selected surfaces};
  only_use_for_effects {selected surfaces wallpapered} = true;
end_schema;

schema install_kitchen;
  expands {install kitchen appliances};
  only_use_for_effects {kitchen appliances installed} = true;
end_schema;
```

```
schema install_water_heater;
  expands {install hot water heater};
  only_use_for_effects {hot water heater installed} = true;
end_schema;

schema install_heating;
  expands {install heating and cooling ducts};
end_schema;

schema install_furnace;
  expands {install furnace and air conditioner};
  only_use_for_effects {furnace and air conditioner installed} = true;
end_schema;

schema complete_trim;
  expands {complete exterior trim};
  only_use_for_effects {exterior trim complete} = true;
end_schema;

schema lay_brickwork;
  expands {lay brickwork exterior walls plus inside fireplace};
  only_use_for_effects {brickwork laid} = true;
end_schema;

schema single_roof;
  expands {shingle roof};
end_schema;

schema attach_gutters;
  expands {attach gutters and downspouts};
  only_use_for_effects {gutters and downspouts attached} = true;
end_schema;

schema paint_exterior;
  expands {paint exterior trim};
end_schema;

schema place_insulation;
  expands {place insulation in attic};
end_schema;

schema lay_walks;
  expands {grade lay forms for walks and driveways};
end_schema;

schema pour_walks;
  expands {pour walks and driveway};
end_schema;

schema finish_grading;
  expands {finish grading};
end_schema;

schema landscape_yard;
  expands {landscape_yard};
end_schema;

schema install_switches;
  expands {install electrical outlets switches lighting fixtures};
end_schema;

schema final_hookup;
  expands {final hookup of electrical system};
end_schema;
```

```
schema clean_up;
  expands {clean up interior and exterior including yard};
  only_use_for_effects {interior and exterior cleaned} = true;
end_schema;

schema carpet;
  expands {lay carpeting};
end_schema;

schema attach_cabinet_fixtures;
  expands {attach cabinet fixtures};
end_schema;

schema close_out;
  expands {close out house};
end_schema;
```

C.2 Full pacifica domain specification

```
Task Operation_Columbus
 nodes sequential
          1 start,
          parallel
           3        action (transport_ground_transports Honolulu Delta)
           4        action (transport_helicopters Honolulu Delta)
          end_parallel
          parallel
           5        action (evacuate Abyss 50)
           6        action (evacuate Barnacle 100)
           7        action (evacuate Calypso 20)
          parallel
           8        action (fly_passengers Delta Honolulu)
           9        action (transport_ground_transports Delta Honolulu)
           10       action (transport_helicopters Delta Honolulu)
          end_parallel
          2 finish
 end_sequenctial;
 effects
          (location_gt GT1)              =  Honolulu at 1
          (location_gt GT2)              =  Honolulu at 1
          (in_use_for GT1)               = in_transit at 1
          (in_use_for GT2)               = in_transit at 1
          (location_at AT1)              = Honolulu at 1
          (in_use_for AT1)               = in_transit at 1
          (apportioned_forces GT)        at 1
          (apportioned_forces AT)        at 1
          (at C141)                      = Honolulu at 1
          (at C5)                                = Honolulu at 1
          (at KC10)                      = Honolulu at 1
          (at B707)                      = Delta at 1
          (runway_status_at Delta)       = clear at 1
          (runway_status_at Honolulu)    = clear at 1
          (gt_capacity 25)                         at 1
          (at_capacity 35)                         at I

end_task;

schema evacuate_city
 expands  {evacuate city ?city ?number}
 vars     ?city                  = ?{type city}
          ?number   = ?{satisfies numberp}
 conditions
          achieve {evac_status ?city} = {0 ?number};
 end_schema;
```

```
schema Road_Transport
 only_use_for_effects {evac_staus ?from} = {e_left e_safe};
 vars ?from =?{type city}
 ?to =?{type air_base}
 ?gt =?{type ground_transport}
 ?e_left =?{type numberp}
 ?e_safe =?{type numberp }
 ?c_left =?{type numberp }
 ?c_safe =?{type numberp }
 ?capacity =?{type numberp }
 ?take =?{type numberp }
 nodes
   1 action {drive ?take in ?gt from ?from}
   2 dummy
 conditions
   only_use_if {apportioned_forces GT}
   only_use_if {evacuate_to ?to}
   only_use_if {gt_capacity ?capacity}
   compute {?capacity ?e_left ?e_safe} = {?c_left ?c_safe)
   compute              {- ?e_safe ?c_safe) = ?take
   achieve {evac_status ?from} = {?c_left ?c_safe) at 2
   unsupervised {location_gt ?ft} = ?to at begin_of 1
   unsupervised {in_use_for ?gt} = available at begin_of 1
   supervised {in_use_for ?gt} = ?from at end_of 1 from begin_of 1
 effects
   {in_use_for ?gt} = ?from at begin_of 1
   {in_use_for ?gt) =  available at end_of 1
end_schema

Schema Air_Transport
 only_use_for_effects {evac_staus ?from} = {e_left e_safe};
 vars        ?from =?{type city}
             ?to =?{type air_base}
             ?at =?{type airtransport}
             ?e_left =?{type numberp}
             ?e_safe =?{type numberp }
             ?c_left =?{type numberp }
             ?c_safe =?{type numberp }
             ?capacity =?{type numberp }
             ?take =?{type numberp }
 nodes
   1 action {fly ?take in ?gt from ?from}
   2 dummy
 conditions
   only_use_if {apportioned_forces AT}
   only_use_if {evacuate_to ?to}
   only_use_if {gt_capacity ?capacity}
   compute  {?capacity ?e_left ?e_safe} = {?c_left ?c_safe)
   compute              {- ?e_safe ?c_safe) = ?take
   achieve {evac_status ?from} = {?c_left ?c_safe) at 2
   unsupervised {location_gt ?AT} = ?to at begin_of 1
   unsupervised {in_use_for ?at} = available at begin_of 1
   supervised {in_use_for ?at}    = ?from at end_of 1 from begin_of 1
 effects
   {in_use_for ?at}    = ?from at begin_of 1
   {in_use_for ?at) = >available at end_of 1
end_schema
```

```
schema transport_ground_transports
  expands {transport_ground_transports ?from ?to}
  vars      ?from                    = ?{type air_base}
            ?to                      = ?{type air_base}
  nodes
    1 action {load ground_transports}
    2 action {take_off_from ?from}
    3 action {fly_to ?to}
    4 action{land_at ?to}
    5 action {unload ground_transports}
  orderings 1->2, 2->3,3->4,4-5
  conditions
    achieve {at c5} = ?from at 1
    unsupervised {location_gt GT1} = ?from at 1
    unsupervised {location_gt GT2} = ?from at 1
    unsupervised {runway_status_at ?from} = clear at begin_of 2
    supervised {runway_status_at ?from} = inuse at end_of 2 from begin of 2
    unsupervised {runway_status at ?to} = clear at begin_of 4
    supervised {runway_status_at ?to} = in_use at end_of 4 from begin of 4
  effects
    {at c5} = ?to at 5
    {location_gt GT1} = ?to at 5
    {location_gt GT2} = ?to at 5
    {in_use_for GT1} = available at 5
    {in_use_for GT2} = available at 5
    {runway_status_at ?from} = in_use at begin_of 2
    {runway_status at ?from} = clear at end of 2
    {runway_ststus at ?to} = in_use at begin of 4
    {runway_status at ?from} = clear at end_of 4
end_schema

schema transport_helicopters
  expands {transport_helicopters ?From ?to}
  vars      ?from                    = ?{type air_base}
            ?to                      = ?{type air_base}
  nodes
    1 action {load air_transports}
    2 action {take_off_from ?from}
    3 action {fly_to ?to}
    4 action{land_at ?to}
    5 action {unload air_transports}
  orderings 1->2, 2->3,3->4,4-5
  conditions
    achieve {at c141} = ?from at 1
    unsupervised {location_at AT1} = ?from at 1
    unsupervised {runway_status_at ?from} = clear at begin_of 2
    supervised {runway_status_at ?from} = inuse at end_of 2 from begin of 2
    unsupervised {runway_status at ?to} = clear at begin_of 4
    supervised {runway_status_at ?to} = in_use at end_of 4 from begin of 4
  effects
    {at c140} = ?to at 5
    {location_gt AT1} = ?to at 5
    {in_use_for AT1} = available at 5
    {runway_status_at ?from} = in_use at begin_of 2
    {runway_status at ?from} = clear at end of 2
    {runway_ststus at ?to} = in_use at begin of 4
    {runway_status at ?from} = clear at end_of 4
end_schema
```

```
schema fly_passengers.
 expands {fly_passengers ?from ?to}
 vars      ?to        = ?{type air_base}
           ?from      = ?{type air_base}
 nodes
   1 action {load Passengers}
   2 action {take_off_from ?from}
   3 action {fly_to ?to}
   4 action{land_at ?to}
   5 action {unload Passengers}
 orderings 1->2, 2->3,3->4,4-5
 conditions
   unsupervised {location_gt B707} = ?from at 1
   unsupervised {runway_status_at ?from} = clear at begin_of 2
   supervised {runway_status_at ?from} = inuse at end_of 2 from begin of 2
   unsupervised {runway_status at ?to} = clear at begin_of 4
   supervised {runway_status_at ?to} = in_use at end_of 4 from begin of 4
 effects
   {at B707} = ?to at 5
   {runway_status_at ?from} = in_use at begin_of 2
   {runway_status at ?from} = clear at end of 2
   {runway_ststus at ?to} = in_use at begin of 4
   {runway_status at ?from} = clear at end_of 4
   {nationals out} = true at 5
end_schema
```

Blank