JOBPLAN — A NEW INTEGRATED REPRESENTATION AND PLANNER FOR BATCH JOB WORKFLOW AUTOMATION

BY TRACEY D. LALL

A dissertation submitted to the
Graduate School—New Brunswick
Rutgers, The State University of New Jersey
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
Graduate Program in Computer Science

Written under the direction of
Matthew Stone
and approved by

______________________
______________________
______________________
______________________

New Brunswick, New Jersey
May, 2011
This dissertation presents a new representation and action logic for integrated planning, scheduling, execution monitoring and sensing. These features were motivated by the problem of computer batch job management but are applicable to any domain entailing these forms of reasoning. The existing planning literature has primarily focussed on providing highly efficient representations and algorithms which address specific aspects of planning and sensing. However no single planning framework currently combines the requisite integrated abilities of managing durative triggered actions in an open world environment. The dissertation’s contributions are a multi-purpose planning and sensing representation and an associated partial order action logic to support these features. Plans and beliefs are represented as a workflow state machine governed by a clearly defined dynamics. Time based goals are handled by treating time as a fluent. The implementation and evaluation of a prototype planner “JobPlan” on key domain scenarios illustrating these features is presented.
Acknowledgements

Matthew Stone for his invaluable guidance, insight and patience, my dissertation committee Alex, Shan and Ron for their diligence and valuable feedback, my understanding boss Emma for her flexibility and Bank of America for their generous tuition support.
Dedication

To Johanna and my parents Pat and Ran for their love, caring and encouragement.
Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>iii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiv</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1. AI Planning Techniques for Batch Job Automation</td>
<td>3</td>
</tr>
<tr>
<td>1.2. Structure of the Dissertation</td>
<td>4</td>
</tr>
<tr>
<td>2. THE NEED FOR A NEW WORKFLOW PLANNING REPRESENTATION</td>
<td>6</td>
</tr>
<tr>
<td>2.1. Planning Primer/Refresher</td>
<td>6</td>
</tr>
<tr>
<td>2.1.1. What is planning?</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2. Domain characteristics</td>
<td>7</td>
</tr>
<tr>
<td>2.1.3. Planning domain description languages</td>
<td>9</td>
</tr>
<tr>
<td>State representations</td>
<td>10</td>
</tr>
<tr>
<td>Action representations</td>
<td>11</td>
</tr>
<tr>
<td>2.1.4. Plan representations</td>
<td>11</td>
</tr>
<tr>
<td>2.1.5. Planning inference mechanisms</td>
<td>12</td>
</tr>
<tr>
<td>Forwards inference</td>
<td>13</td>
</tr>
<tr>
<td>State based backwards Inference</td>
<td>14</td>
</tr>
<tr>
<td>Partial order backwards search</td>
<td>14</td>
</tr>
<tr>
<td>Graph based planning</td>
<td>15</td>
</tr>
</tbody>
</table>
Hierarchical planning .................................................. 15
SAT-planning ............................................................. 16
Policy iteration ........................................................... 16
Theorem proving ........................................................ 16

2.2. Required Planning Features ........................................ 17

2.2.1. Planning with concurrent actions and events which have a delayed effect 17
2.2.2. Completion monitoring of durative actions .......................... 18
2.2.3. Planning for goals with simple temporal constraints ............... 20
2.2.4. Planning under partial observability ............................... 21
    Indirect sensing or diagnosis ........................................... 23
    Existing approaches to sensing .......................................... 24
    Existing approaches to diagnosis ....................................... 26
2.2.5. A compact human readable plan representation .................... 29
2.2.6. Planning with exogenous events which are triggered by the occurrence of particular conditions ..................... 30
2.2.7. Execution of actions whose preconditions have not been verified as holding ................................................. 31
2.2.8. Summary of Features Versus Planners .............................. 34

2.3. Determining the domain requirements ............................... 36

2.3.1. Batch job scheduling ................................................. 36
2.3.2. Existing automation approaches ..................................... 36
2.3.3. Approach for determining the planning and diagnostic features needed to create an automated agent for this domain ............. 37
2.3.4. Case Study ............................................................ 39
    System architecture of the batch job environment ................... 39
2.3.5. Methodology .......................................................... 40
    Expressing the requirements ............................................. 43
3. NEW INTEGRATED PLANNING AND DIAGNOSTIC REPRESENTATION

3.1. Requirements of the Representation

3.2. Formulation of New Representation

3.3. New Plan Representation

3.3.1. Plan execution

3.3.2. Reasoning about the plan execution dynamics

3.3.3. Describing the plan dynamics using state and actions descriptions

3.3.4. Plan generation

3.3.5. Example sequential plan

3.3.6. Example plan with external event monitoring

3.3.7. Example plan with actions with unverified preconditions

3.4. Representation of Beliefs and Sensing

3.4.1. Using plan variables to represent beliefs

3.4.2. Belief representation and establishment

3.4.3. Plan with knowledge acquisition

3.4.4. Plan using external medium to record sensing results — medical problem

3.4.5. Plan to determine combination of a safe by trying different combinations

3.4.6. Plan with contingent knowledge acquisition using external media to record results and plan merge

3.5. Existing Time Representations and How the Plan Representation can Support Temporal Reasoning

3.5.1. Differences between time and other fluents in terms of planning
3.5.2. Time conditions .............................................. 84
3.5.3. Durative events .............................................. 84
3.5.4. Causal orderings and temporal relations ................. 85

4. PLANNING WITH THIS REPRESENTATION ...................... 90
  4.1. Plan Generation from the Transition Function .............. 90
    4.1.1. Models and transition function for reasoning with the JobPlan representation .............................................. 91
    Domain of discourse ........................................... 91
    Transition function ............................................ 92
  4.2. An Action Logic for this Transition Function .............. 95
    4.2.1. Syntax .................................................... 96
    4.2.2. Models .................................................. 96
      Atomic terms ................................................. 98
      Ground state predicates ..................................... 98
      Zero place predicates ....................................... 99
      Trajectory predicates ....................................... 99
  4.3. Axioms ....................................................... 104
    4.3.1. Causality and action effects .............................. 104
      Proof of state occurrence on trajectory .................... 104
      State occurrence given condition occurrences ........... 105
      State condition occurrence from causal support ........... 105
      State condition occurrence from occurrence of antecedent conditions 106
      Event occurrence from occurrence of trigger state ....... 106
    4.3.2. Orderings between states ................................ 106
      Event effect state is ordered after its trigger state .... 107
      All states ordered after current state ..................... 107
      State occurs after all enabling events .................... 107
      Transitive ordering .......................................... 108
Time based ordering .............................................. 109

4.3.3. Proving condition protections ................................ 109
Protection if no threats ............................................. 109
Threat resolved by demotion ....................................... 110
Threat resolved by promotion ...................................... 110
Threat resolved by contingency separation .................... 111

4.3.4. Disproving occurrence of a state on a trajectory .......... 111
Disprove state occurrence from disproof of state condition occurrence 111
Disprove state condition from disproof of supporting event ........ 111
Disprove state condition from ordering of the needed condition till
after needed state .................................................. 112
Supporting events disabled ......................................... 112

4.3.5. Reasoning with sets of trajectories .......................... 113
Occurrence proven for subset of trajectories ....................... 114
Occurrence proven for union of trajectories ....................... 114

4.4. Plan Synthesis .................................................. 114

4.4.1. Completeness ................................................. 117

4.4.2. Determining all causal antecedents ......................... 119

4.4.3. Proof procedure for determineCausalAntecedents ........ 120
To prove a state occurrence on a trajectory ....................... 120
To prove the occurrence of a state condition ..................... 121
To prove an ordering from one event to another ................ 122
To prove a protection from stateA to another stateB .......... 123
To disprove occurrence of a threat .............................. 124
To disprove occurrence of a state ................................ 124
To disprove occurrence of a state condition ..................... 125
To prove a state occurrence on some contingencies and disprove on
others ............................................................. 125
To prove a true belief variable on some contingencies and false on others 126
To disprove occurrence of an exogenous event .................................. 127

4.4.4. Handling contingencies in subgoals ........................................ 128

4.5. Evaluation of the planner implementation ................................. 130

4.5.1. Handling exogenous events, action monitoring and triggered actions 131

4.5.2. Planning with knowledge goals, use of the obtained knowledge and merged plan branches ................................................. 135

4.5.3. Sensing using exogenous action to obtain knowledge and use of an external medium to record knowledge ............................... 153

4.5.4. Contingent planning with temporal goals ................................ 156

5. PLANNER IMPLEMENTATION .................................................. 162

5.1. Planner Design ................................................................. 162

5.1.1. Choice of language .......................................................... 162

5.1.2. Implementation of generatePlan(goal) .................................... 164

5.1.3. Implementation of proveGoalOnUC(PC, UC, Plan, goal) ........... 166

5.1.4. Implementation of determineCausalAntecedents(UC, Plan, SG) . 166

5.1.5. Performing forwards inferences to show the plan achieves the goal . 167

5.1.6. Logical retraction, inconsistency checking and backtracking ....... 168

Logical retraction ................................................................. 168

Inconsistency detection .......................................................... 169

Backtracking ................................................................. 170

5.1.7. Inference control ............................................................. 170

Prevention of circular contingency equivalence inferences .......... 170

Reducing redundant deductions about occurrences ................. 172

Workaround for limitations of drools logical retraction mechanism . 173

Prevention of multiple rule fires ........................................ 174

5.1.8. Example rules .............................................................. 174

Example deductive rule .................................................. 174

Example abductive rules .................................................. 177
List of Tables

2.1. General Plan Search Algorithm ............................................. 13
2.2. Required Planner Features .................................................. 17
2.3. Example percepts ................................................................. 22
2.4. Procedural plan equivalent of policy ....................................... 30
2.5. Existing Planner Feature Support .......................................... 35
2.6. Key example scenarios from case study .................................... 41
3.1. JobPlan Syntax ................................................................. 47
3.2. Example JobPlan ................................................................. 48
4.1. Model Based Planner Search Algorithm ................................... 91
4.2. Action Logic Syntax ............................................................ 97
4.3. Top level plan generation algorithm ....................................... 115
4.4. proveGoalOnUC procedure ................................................... 116
4.5. determineCausalAntecedents procedure .................................... 117
4.6. Start, Successful and Failure Event definitions for "genReport ?date". . . . 132
4.7. Start, Success, and Failure event definitions for "ftpToRemote ?file" . . . 132
4.8. Event definition for exogenous event externalFileGen .................. 133
4.9. Event definition for "assign ?x ?y" ......................................... 135
4.11. Start, Success and Failure Event definition for “repairDB i_dbState”. . . 136
4.12. Event definition for DBADiagnose ......................................... 153
4.13. Event definition for emailDBA“requestDiagnosis” ........................ 154
4.15. Event definition for exogenous event setEODMarker .................. 157
4.16. Start, Successful and Failure Event definitions for “runEODReport” . . . . 157
4.17. Start, Successful and Failure Event definitions for "communicateLateSLA". 158

6.1. Type and number of inferences in the proof of plan correctness for the repairDB example .......................................................... 184

A.1. Problem Scenarios for Example System ........................................... 191
A.2. Problem Scenarios for Example System — continued ...................... 192
A.3. Problem Scenarios for Example System — continued ...................... 193
A.4. Problem Scenarios for Example System — continued ...................... 194
A.5. Problem Scenarios for Example System — continued ...................... 195
List of Figures

2.1. Case Study System .................................................. 38
3.1. Diagrammatic convention ........................................... 56
3.2. Sequential Plan — initial state .................................... 57
3.3. Sequential Plan — executing 1st action ............................ 57
3.4. Sequential Plan — 1st action completed ........................... 58
3.5. Sequential Plan — 2nd action executing ........................... 58
3.6. Sequential Plan — execution complete ............................ 60
3.7. Event monitoring plan — initial state ............................. 60
3.8. Event monitoring Plan — exogenous event generates input file ............................ 61
3.9. Event monitoring Plan — report generate job starts ............. 61
3.10. Event monitoring Plan — report generate job completes ......... 63
3.11. Bomb dunk — initial state. ........................................ 63
3.12. Bomb dunk — bomb defused. .................................... 68
3.13. Sensing and repair plan — initial state. .......................... 68
3.14. Sensing and repair plan — sensing action starts. ............... 69
3.15. Sensing and repair plan — sensing action complete, plan variable populated with results of sensing action. .............................. 69
3.16. Sensing plan — repair action starts with the parameter value of 3. ........... 70
3.17. Sensing and repair plan — repair action completes, db repaired and dbState = 0. ........................................... 70
3.18. Medical diagnosis of infected patient using external media — initial state. .... 72
3.19. Medical diagnosis of infected patient using external media — stain action executed ........................................ 73
3.20. Medical diagnosis of infected patient using external media — inspect action executed ................................................................. 73

3.21. Medical diagnosis of infected patient using external media — infected plan variable assigned .................................................... 76

3.22. Safe combination plan — initial state. .................................................. 76

3.23. Safe combination plan — checkCombo actions run ............................ 77

3.24. Safe combination plan — combination value plan variable set .......... 77

3.25. Contingent plan — db not corrupted, initial state. ............................. 80

3.26. Contingent plan — db not corrupted, i_isCorrupted assigned a value of False 80

3.27. Contingent plan — Support alerted that DB is not corrupted ............. 81

3.28. Contingent plan — db corrupted, initial state. ................................... 81

3.29. Contingent plan — db corrupted i_isCorrupted assigned a value of True . 87

3.30. Contingent plan — job run to record corrupted id to log file ............... 88

3.31. Contingent plan — job run to read corrupted id from log file ............ 88

3.32. Contingent plan — Support alerted that DB is corrupted and provided the corrupted id .............................................................. 89
Chapter 1

INTRODUCTION

This dissertation presents a new plan representation, action logic and prototype planner implementation. These were motivated by the following planning capabilities required for the domain problem of automating computer batch job workflows:

- Planning with concurrent actions and durative events whose effects take place at some finite time after the event has been initiated.
- Completion monitoring of durative actions and external events.
- Planning for goals with simple temporal constraints.
- Planning under partial observability.
- Sensing for direct and indirect knowledge goals and subsequent use of the obtained knowledge.
- A compact human readable plan representation.
- Planning with exogenous events which are triggered by the occurrence of particular conditions.
- Execution of actions whose preconditions have not been verified as holding (Morgenstem, 1987; Golden, 1997)

Planning and diagnosis are extensively developed AI fields with a large body of work. Specialised and highly efficient representations and algorithms exist to address each of these individual requirements. These techniques (discussed in Chapter 2) include partial order planners, execution controllers, temporal planners, modal knowledge logics and diagnosis algorithms.
The contribution of this dissertation is to address the domain requirements in a single integrated workflow style plan representation. Planning with this plan representation is performed using a new action logic designed for reasoning about the combined evolution the world and this plan.

One of the advantages of representing and reasoning about these different aspects using a unified representation and associated action logic is that interactions between the different planning requirement are handled. For example planning for a goal which involves a time constraint takes into account the variable duration of a sensing action; an exogenous event is triggered and then monitored for completion; knowledge of a fact is established but only under certain required circumstances.

The created JobPlan plan representation follows the plan as program approach as employed by H. J. Levesque, Reiter, Lesperance, Lin, and Scherl (1997). Within this paradigm a plan is represented as a form of program which is interpreted at runtime and converted into action executions. In the JobPlan representation this program is a workflow style language. Each action in the plan is executed as soon as its defined start conditions become true. These start conditions can reference the completion of previous actions within the plan, the value of internal plan variables (see below) or the value of world fluents. These start conditions may be established by previous plan actions or by exogenous events. The JobPlan representation for sensing is achieved by storing the results of sensing actions in internal variables which form part of the plan. Subsequent plan actions may access these previously populated plan variables for the action parameters. Contingent execution of actions is achieved by including conditions on these plan variables in an action’s start conditions. These plan variables constitute the planner’s knowledge about those attributes of the world which are not directly observable. By reasoning about the state of these variables and their relationship to fluents in the external world, all planning about knowledge is achieved via standard causal reasoning (without the need for use of modal logic). The combined evolution of the workflow plan and world are modeled using a new partial order action logic. The action logic is designed to model triggered events and exogenous events and handles durative events and time based goals and reasoning about contingencies.
The set of required planning capabilities for this domain was captured by a real world case study (described in Chapter 2) on the automation of computer batch job workflow schedules. The test examples in this dissertation were taken from this problem domain. However the representation is general and applicable in any domain which shares this set of requirements.

1.1 AI Planning Techniques for Batch Job Automation

Enterprise computer environments usually involve the scheduled running of hundreds of computer batch jobs, programs and processes on a collection of machines. Automated schedulers exist for the timed launch of these batch jobs and allow job execution to be predicated upon the occurrence of particular conditions. The job schedules may stipulate that alerts to the support team are generated when non-nominal situations arise. Based on their knowledge of the behaviour and resource requirements of each job, schedule definitions must be hand-coded by the support staff, and when alerts occur the staff must at runtime perform the requisite manual problem diagnosis and error correction. This requires a large number of well trained support staff (Murch, 2004; Goldman & Baral, 1998) and the cost of system down time for many enterprises may be considerable (Brown, 2001). The current rule of thumb is that for every dollar spent on computing infrastructure, between 2-10 times that amount are spent for ongoing management and this ratio only increases as system complexity grows. These activities are ripe for further automation using techniques of planning, scheduling and diagnosis from the AI literature. A computer environment presents a real-world opportunity where useful results are amenable to an automated approach:

- Computer environments are typically more deterministic in their behaviour than many other real-world environments and all components of the system have been designed with a recognized form and purpose, hence the dynamic behaviours of the system are known apriori.

- Implementation of interaction with a software environment is much easier to and less costly to achieve than for a physical environment (Etzioni (1993) discusses the attractiveness of software environment as testbeds for AI research with respect to
ease of implementation and the applicability of planning techniques to real-world problems.)

- Testing and evaluation of an approach is feasible due to the controllability and reproducibility of the environment.

Previous automated approaches (Ennis, 1986) for this domain have utilized a pattern based rule approach to error situation identification. The disadvantage of this approach is that rules must be hand-coded by a skilled operator for each computer environment. In order to avoid the need for hand-coding an agent must automatically generate batch job schedules which monitor progress, plan for resource usage and detect, diagnose and correct problems. These schedules must take into account the interactions between the primary components in the computer environment. The agent’s representation of the computer environment must be such that a user untrained in artificial intelligence techniques can input all the information required for the agent to generate the schedules.

1.2 Structure of the Dissertation

The dissertation is structured as 5 additional chapters:

- Chapter 2 describes how each of the required planning features are handled (or not handled) by existing planning and diagnostic representations. These features are illustrated using example scenarios from the real world problem domain. The conclusion is drawn that existing representations have many of the required individual features but none provide all the features in a single integrated representation.

- Chapter 3 presents the new JobPlan representation of a plan as a form of program and illustrates how the formalism is able to support each feature using this integrated representation.

- Chapter 4 describes the action logic created to reason about this plan representation and how generated plans can be proven to be correct with this action logic. Abductive inferences required for plan synthesis are described.
• Chapter 5 describes the implementation of the prototype JobPlan planner using the logic inferences. The performance of the planner on key scenarios from the problem domain is described.

• Chapter 6 discusses the strengths of this representation and action logic, its limitations and suggests future extensions and development possibilities.
Chapter 2
THE NEED FOR A NEW WORKFLOW PLANNING REPRESENTATION

The first section of this chapter provides a brief overview of planning, defining some of the different domain properties which dictate the required forms of planning. An overview of the current space of planner designs in terms of their domain description languages, plan representations and plan generation algorithms is provided. For a discussion of planning techniques see Russell and Norvig (1997); D. Weld (1999); Nau, Ghallab, and Traverso (2004). The second section illustrates each required planning feature with example scenarios from the batch job automation problem domain. Existing planning representations and algorithms from the literature which address these features are discussed. The conclusion is drawn that existing planners have many of the required individual features but do not address all the features using a single integrated representation and reasoning approach, the key issue being that planners which handle sensing do not handle durative actions and vice versa. The third section of this chapter provides an introduction to the problem domain and how the features were identified from the case study.

2.1 Planning Primer/Refresher

2.1.1 What is planning?

A plan consists of a description of which actions to execute and when to execute them. This description should be straightforward to interpret and to convert to a decision on which action to execute at any point during plan execution. Planning is the act of formulating a plan which will transform the initial state of the world into a given goal state which meets some specified criteria. In order to do this the planner requires a domain description
language for describing the planning problem — the state of the world, the goal state and the set of available actions. The preconditions and effects of each action must be described. A planner also requires a search algorithm to find a plan (if any) which solves the given problem. The state of the world is usually described in terms of fluents which are the values (boolean or valued) of properties of the world which may change over time. These may be considered as functions which take a situation as their argument and return a situation-dependent value. If there is uncertainty about the initial state or about the effects of actions, and the goal needs to be proven for all contingencies the created plan must describe which actions should be executed for each contingency.

2.1.2 Domain characteristics

The choice of planner is dictated by the characteristics of the environment. For environments which conform to the closed world assumption the planner has complete information about the entire state of the world and if the planner has no explicit knowledge about a predicate it may be assumed that the predicate is false. In some environments it may be assumed that the effects of actions are deterministic — i.e if the preconditions of the action are met, then the action will have exactly the effects that are specified for that action. The assumption may also be made that the only changes in the state of the world are due to the planner actions. Planning domains which conform to all three of these assumptions are classical planning problems.

In environments where these assumptions are relaxed, the planner must have additional capabilities. Agents which operate under conditions of partial observability need to make the open world assumption that the state of some parts of the world must be modeled as unknown. Any agent operating within a partially observable environment needs to perform observations (or percepts) on the world in order to know the value of relevant fluents and predicates. Some observations are cheap and fast and may be performed automatically whereas other may take significant time or resources and need to be treated as actions and only executed as needed. Such actions are called sensing actions. Sensing actions generally do not change the value of the fluent, they only change the agent’s knowledge of the fluent. Sensing actions must be treated differently from regular actions because of this.
For directly sensable fluents the agent is able to determine the value of the fluent from the result of a single sensing action. For indirectly sensable fluents, there are no sensing actions which directly determine the value of the fluent. The only way the value of such fluents may be determined is indirectly via inference from its causal relationship with other observable fluents. These causal relationships may be either pre-existing static constraints (see below) or relationships established between fluents by actions whose effects are conditional and dependent upon the non sensable fluent. Such conditional actions establish correlations between those non sensable fluents involved in the action preconditions and the sensible fluents established by the action effects. In order to determine the value of indirectly sensable fluents, therefore, inferences and/or multiple sensing actions and inferences from the results of those actions are needed. Diagnosis is the act of performing these requisite sensing actions and inferences. Abductive inference is that form of inference used to derive explanations (using states, or event narratives) of observations and past events.

The side effects of actions as well as their direct effects may also need to be taken into account (the ramifications of actions). A domain may have invariant constraints (domain constraints) which are always in effect. These static constraints determine the side effects of an action. An example of a static domain constraint is that any item on a tray has the same location as that tray. Through consideration of this domain constraint and the fact that a cup is on the tray, a planner should reason that the action of moving the tray to a location will also have the effect that the cup is in that location. This side effect is called the ramification of the action under the domain constraints. In most planning approaches the STRIPS assumption is made that fluents which are not mentioned in the postconditions and which are not affected through ramifications remain unaffected by an action. The frame problem (Shanahan, 1997b) is the problem of how to reason succinctly about those aspects of the world which are not affected by the action via postconditions or ramifications. The frame problem is mainly concerned with how to represent this inertial assumption consistently and concisely within the language of predicate logic.

Non-deterministic actions are those where a particular outcome is not guaranteed even if the preconditions are met, i.e. different outcomes are possible given the same preconditions. Strong planning is the construction of plans which are able to achieve a goal even when there
is uncertainty in the initial state or actions have non-deterministic outcomes. Strong plans may require different actions to be performed under different contingencies and may require cyclical application of the same action. *Conformant planning* is strong planning under the condition of null observability, where no fluents may be sensed. For non-deterministic actions the *qualification problem* is the problem of precisely specifying all of the conditions which must be met in order for an action to have the expected effect. For some real-world actions, there may be effectively an infinite number of things that could go wrong which might prevent an action having its desired effect (e.g. to start a car, it’s not necessarily enough to turn the key in the ignition, theoretically it is also necessary to confirm the tail pipe isn’t blocked, confirm the connectivity in all of the ignition systems etc, etc). Solutions to the qualification problem include the creation of a *possible* fluent which determines if an successful action is possible and where this fluent is a (possibly non deterministic) function of other fluents.

### 2.1.3 Planning domain description languages

One of the original domain description languages was STRIPS (Fikes & Nilsson, 1971). This domain description language uses predicate calculus to describe the initial state of the world and the goal state. A *fluent* is a predicate or function used to model some aspect of the world which may vary with time, e.g. *color(block,green)* which denotes that a block is green. STRIPS defines an action by defining its preconditions and postconditions. The preconditions are expressed as a set of atomic predicates which describe those aspects of the world state which must be true in order for the action to be executed — e.g. an action which pours tea from a pot to a cup would have the predicate precondition *empty(cup)*. The postconditions of the action are expressed as a set of atomic predicates which will be true after the action has been executed (the “add” list for the action.) and a set of atomic predicates which will no longer be true after the action has been executed (the “delete list”). In the tea cup example the add list would contain the formula *full(cup)* and the delete list would contain the formula *empty(cup)* since the cup is no longer empty after the action has been executed. The STRIPS domain description language is designed for environments which conform to the classical planning assumptions. ADL (Pednault, 1989) is a richer
domain description language which extends the STRIPS language through allowing the actions to have preconditions involving disjunction, negation and quantification. ADL also supports context dependent actions where the action may have different outcomes depending on the conditions which hold when the action is executed. PDDL (McDermott, 1998) is a standardized domain description language which has been used to describe test problems for planning competitions. It is a superset of ADL and STRIPS. PDDL has become the de-facto standard for most modern planning systems and has evolved to include support for additional features in different versions. PDDL 2.1 allows use of real-valued fluents and preconditions using inequalities between those fluents and the modeling of action durations and goals which involve time values. PDDL 2.2 supports the concept of *timed initial literals* which define the value of fluents which hold at particular time points due to the effects of exogenous events. PDDL 3.0 extends PDDL 2.2 by trajectory constraints and preferences. A *trajectory constraint* is a constraint on the set of valid plans; they are expressed in a temporal logic. *Preferences* allow expressing soft constraints (soft trajectory constraints, soft preconditions and soft goals), which are constraints that need not be satisfied by a plan, but lead to a decrease in plan quality if they are not. NuPDDL (Bertoli, Cimatti, Lago, & Pistore, 2003) extends PDDL in that it allows non-deterministic actions and strong planning under partial observability.

**State representations**

Ground state representations explicitly define the value of every fluent. For partially observable environments, a belief state may be described in lieu of the state of the world (which is not accessible). Symbolic representations use boolean connectives to allow sets of ground states to be represented succinctly without explicitly listing each ground state. Symbolic representations such as binary decision diagrams used by the MBP planner (Bertoli, Cimatti, Pistore, Roveri, & Traverso, 2001) are able to efficiently reason about large sets of states. Knowledge based approaches encode the world state indirectly in terms of the knowledge state of the planner. This knowledge state provides a symbolic representation of the possible worlds which the planner is in. State representations in general may or may not encode the time value as part of the state. Some planners represent time separately
using a temporal database of states which either hold at a particular point in time or over an interval of time.

**Action representations**

During plan construction in order to decide which actions to place into the plan (and under which contingencies those actions need to occur) the planner must reason about the effects of actions on a state. At a minimum, actions must be defined in terms of the conditions which must hold in order for the action to be executable and the conditions which hold after the action has been executed (i.e. the effects of the action). Most planners employ a STRIPS-like representation with the preconditions and postconditions being stipulated using predicate formula. For actions with conditional effects, there may be multiple precondition-postcondition definitions for a single action, defining its behaviour under different circumstances. For state representations based on planner knowledge of the world, actions may be defined in terms of their preconditions and effects upon the agent’s knowledge rather than upon the state of the external world. For actions which are nondeterministic, the probability of different possible action outcomes may also be specified. For actions which have a time duration this may need to be represented for each action.

### 2.1.4 Plan representations

A plan representation must contain sufficient information for the correct sequence of actions to be executed at runtime. Various representations exist to encode this information. Planners (such as PKS (R. Petrick & Bacchus, 2002), FF (Hoffmann & Nebel, 2001), MBP (Bertoli et al., 2001)) use a tree-based representation to represent contingent plans. Each branch in the tree corresponds to a different outcome of a sensing action. Each path from the root of the tree to a leave defines a unique sequence of actions to be executed given a particular series of sensing outcomes.

*Partial order* planners such as Puccini (Golden, 1998) and Cassandra (Pryor & Collins, 1996) represent the plan as a partially ordered set of actions. The orderings are represented as a graph where each node represents an action and the edges represent dependencies on other actions. An action is executed after its dependency actions have been executed. If
the graph contains two actions $A$ and $B$ which are not ordered with respect to each other in the graph then executions which execute $B$ before $A$ and executions which execute $A$ before $B$ are both valid. Contingent plans may be represented with a partial order graph by introducing branches into the graph if there is uncertainty in the precondition of an action (as used by Cassandra). Each node is labeled with the contingencies on which it occurs.

Program representations (such as the Golog planning language (H. J. Levesque et al., 1997)) may be used to describe the plan in terms of a composition of action execution statements and standard program constructs such as $if$ $then$ (needed for conditional plans) and $do$ $while$ (for iterative plans). Such a representation must be interpreted by a runtime interpreter to convert it into a sequence of concrete action executions.

A plan may be represented as a policy $\pi$ which is a function which maps a state to an action. This representation is used by Markov decision based planners (such as mGPT (Bonet & Geffner, 2005)). This representation can handle contingent executions without the need for contingency labeling schemes since the contingent execution of an action $a$ can be encoded by ensuring that all states $s$ for which $\pi(s) = a$ conform to the contingency on which that action is needed.

### 2.1.5 Planning inference mechanisms

Regardless of the action, plan or state representation formulating the plan boils down to some form of search — searching for a plan of actions the overall effect of which will bring about the goal state. At a high level the general search algorithm can be summarized by the algorithm shown in Table 2.1. The plan is a solution if the plan ensures that under that plan the world will evolve into a state which meets the goal definition. The operation $\text{choose}$ designates a non-deterministic choice of plan modification — the search algorithm must implement backtracking for this branch point. The available plan modifications may include the addition of new actions, introduction of orderings between actions and modifications to plan control structures, depending on the plan representation used. Different search strategies are employed by the different planners.
Table 2.1: General Plan Search Algorithm

\[
\begin{array}{ll}
\text{function} & \text{search}(\text{initial, goal, actions}) \ \text{return} \ \text{plan} \\
\text{plan}=\text{empty} \\
\text{found}=\text{false} \\
\text{while} \ \text{found} \neq \text{true} \\
& \text{if} \ \text{plan} \ \text{is a solution} \ \text{then} \\
& \quad \text{found}=\text{true} \\
& \ \text{else} \\
& \quad \text{choose} \ \text{plan} \ \text{modification} \ (\text{with necessary contingency control if needed}) \\
& \ \text{end} \\
\text{end} \\
\text{return} \ \text{plan} \\
\end{array}
\]

**Forwards inference**

Forwards searching planners start with the initial state and plan forwards. Initially applicable actions are those actions all of whose preconditions hold in the initial state. The planner adds an applicable action into the plan and the effects of the action applied to the initial state to obtain the successor state (by adding the predicate atoms from the action add list and deleting the predicate atoms from the action delete list). With the assumption that the effect of actions is immediate, (there are no concurrent actions) the successor state (or set of states if the action is not deterministic) can be determined by applying the effects of the chosen action to the current state. Rather than choosing the action arbitrarily from the applicable actions, the planner may employ heuristics to decide on which action to add to the plan. The planner chooses the action which results in a state which achieves the lowest value of a heuristic. A heuristic is a function which approximates the distance from a given state to a goal state. An example distance measure would be the shortest plan which starts with the given state and results in the goal state. For example the heuristic function used by FF (Hoffmann & Nebel, 2001) is the length of the shortest solution plan from current state to goal state achieved by applying graphplan (Blum & Furst, 1995) to a relaxation of the planning problem. (The relaxation of the problem is achieved by ignoring the delete list of the actions).
State based backwards Inference

Backwards searching planners start with the goal state and regress backwards until the initial state is reached. Initially applicable actions are those whose postconditions match one or more conditions of the goal state. As actions are added to the plan, the applicable actions become those whose postconditions match one or more conditions of the goal state or which match one or more unsatisfied preconditions of actions already added to the plan. As with forwards inferencing, with the assumption that the effect of actions is immediate, (there are no concurrent actions) the predecessor state (or set of states if the action is not deterministic) can be determined by deleting the predicates from the action effect add list and adding the predicates from the action effect delete list and action preconditions. The regression continues until the regressed state is equal to the initial state, or in the case of contingent planning the regressed state set is a superset of the initial state set. If the state transition is not deterministic then the regressed predecessor state set must contain only those states which are guaranteed to transition to a state in the current state set. Model based planning uses state based backwards inference. It models the world as a non-deterministic finite state machine using binary decision diagrams (BDDs) to efficiently represent the state sets. The backwards state search is performed using the BDD’s to efficiently construct the AND-OR graph of possible predecessor actions which are guaranteed to evolve into the current state.

Partial order backwards search

With a partial order plan it is not possible to calculate the entire predecessor state by un-applying each action in sequence because the exact sequence of actions is not specified in this representation. Instead the planner performs regression in terms of the individual preconditions of actions. The planner maintains a list of subgoals (which are goal conditions or preconditions of actions within the plan) which are not yet met by either the initial state conditions or the postconditions of actions within the plan. The planner inserts actions into the plan whose effects establish an outstanding subgoal conditions. If a subgoal is satisfied by the effects of an action (the supporting action) the planner must ensure that no other actions interfere with that support. If action A has a postcondition which meets the
precondition of action \( C \) and a post-condition of action \( B \) implies the negation of action \( C \)'s precondition, then action \( B \) is said to \textit{threaten} the causal support from action \( A \) to \( C \). Such threats need to be \textit{resolved} by the planner either through ensuring that action \( B \) is executed before action \( A \) (\textit{promotion}), or by ensuring that action \( B \) is executed after action \( C \) (\textit{demotion}). If the environment is one where there is uncertainty in the initial state, the threat posed by \( B \) can also be resolved by ensuring that \( B \) does not occur on those contingencies where \( C \) is needed. By only specifying a partial order on the actions, the size of the search space of plans is much smaller than the space of totally ordered sequential plans.

**Graph based planning**

Graph based planners such as graphplan (Blum & Furst, 1995) and sensory graphplan (D. S. Weld, Anderson, & Smith, 1998) use a graph representation where the nodes in each layer of the graph consist of propositions (even numbered layers — initial layer being zero) and actions (odd numbered layers). The search consists of two phases. The first proceeds forward considering potential actions at each layer of the plan and eliminating invalid solutions via the detection of ‘mutex’ relations — actions that are inconsistent with respect to one another in terms of their preconditions or effects. The second phase traverses the graph produced by the first phase and performs backwards reasoning to choose an action from each level of the graph in order to address the subgoals at each level.

**Hierarchical planning**

Hierarchical planning (such as that used by the SIPE planner (Wilkins, 1997), SHOP (Nau et al., 2003)) attempts to reduce the planner search space by representing and reasoning about some actions at a higher level when constructing the top level plan. Some ground level actions are grouped into higher level composite tasks. The preconditions for the composite task are the preconditions of all the underlying actions except those furnished by other actions within the group. The post conditions for the composite task are the postconditions of all the underlying ground actions — except those which are disabled by another action within the group. The planner first reasons with the higher level tasks.
Once a consistent plan has been found with the higher level tasks, the plan is then refined to ground level actions by expanding out the high level tasks. By first finding a consistent plan with the higher level tasks the planner can reduce the size of its search space.

**SAT-planning**

For goal and action representations which are written purely in terms of propositional fluents, the planning problem can be encoded as an instance of boolean satisfiability. This is the problem of finding an assignment to boolean variables such that a given boolean expression involving those variables is true. Although the SAT problem in its general form is NP-complete (as is STRIPS-planning), efficient algorithms exist which can provide reasonable performance for many planning problems (Kautz, 1992)

**Policy iteration**

For Markov decision based planners, the policy is obtained via a dynamic programming algorithm which estimates a heuristic for the optimal policy. Via an iterative approach the accuracy of this estimate is improved with each iteration and eventually the algorithm converges on the optimal policy. Variations of policy iteration algorithms have been created to handle partially observable domains (Geffner & Bonet, 1998), concurrent actions (Mausam & Weld, 2004) and relational fluent state descriptions (Boutilier, 2001)

**Theorem proving**

For theorem proving based planners plan generation is the process of choosing a plan such that occurrence of the goal may be proven within an action logic describing the dynamics of the domain. The axioms of the proof include the facts known about the initial state of the world and facts known about the plan. During plan construction the theorem prover makes abductive choices about the actions contained in the plan in order to try and prove the goal occurrence theorem. The event calculus abductive planner (Shanahan, 2000) is an example of a theorem proving planner which uses the event calculus action logic.
2.2 Required Planning Features

Table 2.2 shows the required features for a planner which were gathered from the problem domain.

Table 2.2: Required Planner Features

<table>
<thead>
<tr>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Planning with concurrent actions and events which have a delayed effect.</td>
</tr>
<tr>
<td>2. Completion monitoring of durative actions and external events.</td>
</tr>
<tr>
<td>3. Planning for goals with simple temporal constraints.</td>
</tr>
<tr>
<td>4. Planning under partial observability.</td>
</tr>
<tr>
<td>5. Sensing for direct and indirect knowledge goals and subsequent use of the obtained knowledge.</td>
</tr>
<tr>
<td>6. A compact human readable plan representation.</td>
</tr>
<tr>
<td>7. Planning with exogenous events which are triggered by the occurrence of particular conditions.</td>
</tr>
<tr>
<td>8. Actions whose preconditions have not been verified as holding (Morgenstem, 1987; Golden, 1997)</td>
</tr>
</tbody>
</table>

The following sub-sections discuss how each of these required features arises in the problem domain and how such a feature is addressed by existing planners. The conclusion is drawn that a new planner is needed to address the full set of features in an integrated manner.

2.2.1 Planning with concurrent actions and events which have a delayed effect

Within this planning domain an action (for example running a database maintenance job) may take many hours to run and the runtime may vary each time the action is run. The planner must therefore reason about actions whose effects do not occur immediately. During the running of this job the planner may need to run other jobs which may complete before the database maintenance job completes. The planner therefore needs to be able to reason
about actions which execute concurrently.

The planner also needs to handle actions with variable and non-deterministic duration, this means the ordering of the effect of actions does not correspond to the order in which the actions are initiated. For a large number of actions reasoning about all possible orderings of effects is not tractable unless partial order reasoning is used for planning. Many planning approaches assume that the effects of action are immediate, this includes: model based planners (Bertoli et al., 2001), Markov decision based planners (Bonet & Geffner, 2005), satisfiability based planners (Kautz, 1992), graph based planners (Blum & Furst, 1995), and forwards based planners such as PKS (R. P. A. Petrick, 2006). Such planners will need to construct fully ordered plans which handle all possible inter-leavings of actions effects which are consistent with the apriori bounds on durations. To reason efficiently about actions with variable duration a partial order approach to plan representation and reasoning must be used. Partial order planners include SIPE (Wilkins, 1997), Puccini (Golden, 1997) and Prodigy (Veloso et al., 1995). Logic-based planners may or may not employ partial order representations, depending on the particular action logic. The event calculus has variants which support durative actions. (Mueller, 2006) though the event calculus planner ‘Abdemo’ described in Shanahan (2000) does not handle such a variant. Dialects of the plan language Golog exist which can handle durative actions (as activities which hold over an interval of time) (Finzi & Pirri, 2004) and which solve the temporal constraints using constraint solvers.

2.2.2 Completion monitoring of durative actions

Besides reasoning about different inter-leavings of the effect of actions, an approach for monitoring and determining when an action or exogenous event is complete during plan execution is required (since the effect of a planner action may occur some variable time after the action or event has been initiated). For agent actions it may be assumed that the status of the agent action execution is known by the planner at runtime. However for exogenous events, the external event needs to be monitored for completion (since the planner should not execute any actions which rely on the effects of this exogenous event until the exogenous event is completed). For example in order to determine whether an
an exogenous event which generates an input file is complete (perhaps a remote computer batch job not under control of the planner), at a minimum it would be necessary to monitor for the existence of the file.

Either such monitoring is performed by the (online) runtime plan executor, or monitoring directives are include in the plan by the (offline) plan generator. The approach taken by most planners is the former. Some planners (Bresina & Washington, 2001) have sophisticated execution controllers which not only monitor when an action is completed, but also monitor for unexpected results of actions and resource constraint violations. For highly dynamic environments (for example a robot environment) execution controllers may employ iterative control loops to tightly sense and react to changes in the environment in order to maximise the chance of a successful action. For example the Kmonitor execution controller (Eiter, Fink, & Senko, 2005) performs periodic checks on whether the latest state is on a trajectory to the goal state. If not, the execution monitor invokes a diagnostic procedure to determine the cause of the deviation. With such an execution monitor the planner then must invoke a pre-built alternative plan or perform plan deliberation to modify the existing plan to handle the new situation. (Ambros-Ingerson & Steel, 1990) performs interleaved action execution and planning. If during plan execution the world state is changed by events outside of the planner control and a future action in the plan has support for one of its preconditions invalidated by the world state change, then a ‘flaw’ is raised. (A flaw is a standard partial order planning terminology which describes an outstanding requirement, inconsistency or unexecuted part of the plan). When a flaw arises during execution the the planner then makes the necessary plan changes to remove the flaw, by achieving support for the precondition via another means. The Golog planner (H. J. Levesque et al., 1997) supports a construct while, do wait which can be used to monitor when a given condition changes.

For the batch job domain, for scalability to hundreds or thousands of jobs, a simple “lightweight” plan executor is needed. This can be achieved by including the necessary monitoring control directives apriori in the plan by specifying the detectable conditions indicating when an action can be executed. With this the plan executor is very simple — simply requiring monitoring for these conditions at the time of plan execution.
2.2.3 Planning for goals with simple temporal constraints

Existing computer batch job schedulers UC4 (UC4Corp, 2002), Autosys (ComputerAssociates, 2002), Opalis Robot (Opalis Software Inc., 2004) are capable of executing computer batch job runs according to a pre-defined schedule and model basic dependencies between batch jobs (e.g. job B depends on job A and should not start until job A is complete). A job schedule must take into account any deadlines on the deliverables of jobs — for example if a report must be delivered to an external system by 7pm, and the job which generates the report takes up to an hour to run, then the planner needs to schedule the job before 6pm.

Many existing planners have been designed to handle temporal reasoning and specialised temporal logics (Bellini, 2000; Allen, 1991) have been developed for reasoning about different forms of time constraints. The Prodigy planner (Veloso et al., 1995) is able to perform temporal reasoning by treating time as a numeric fluent and handles time based preconditions and effects of actions (Rodriguez-Moreno, Borrajo, & Meziat, 2004). The TLPlan planner (Bacchus & Ady Winter, 2001) is a forwards based classical planner which can handle actions with duration, concurrent actions (by tagging states with timestamps) and metric effects. Simple temporal network (STN) planners (such as those described in Penberthy and Weld (1994); Doherty and Kvarnström (1999)) use a specialised temporal graph representation where the nodes represent temporal events and edges correspond to constraints on the duration between the events. Edges are labelled with the upper and lower bounds of the event durations. A solution to the temporal network is an assignment of time points which adhere to these constraints. The standard STN assumes that the execution time of planner actions are under the control of the planner.

For this domain temporal planning must be combined with contingent planning. Actions may have different durations depending on the value of a fluent which might not accessible to the planner (for example a job with a bad internal state may run for a longer period than an action with a nominal internal state). If the planner cannot guarantee that the internal state is good, it may need to construct a contingent plan which takes into account the two possible durations. Additionally, the planner may need to construct plans where different
actions may need to be executed under different contingencies (for example sensing and repair actions may only need to be executed under contingencies where a batch job fails). In order to meet a particular goal or subgoal time constraint under all contingencies, the timings of actions under those different contingencies will need to be taken into account. Planners which combine contingent planning with temporal planning are a relatively recent area of focus. The PROTTLE planner (Little, Aberdeen, & Thibaux, 2005) addresses actions with probabilistic outcomes and durations using a graph based structure where a node consists either of an action or the outcome of a probabilistic event. When extending the graph the planner can either add a new action or add a wait for the advancement of time and the outcome of an event. Gerevini, Saetti, and Serina (2006) offers an overview of the latest representations and techniques.

2.2.4 Planning under partial observability

In practice neither a human operator nor an agent will know the state of the computer environment at the most detailed level of description (e.g. the byte-level contents of its memory and hard disk). This is either due to limited percepts available for certain aspects of the machine state or for practical feasibility issues. E.g. a percept on a single file is fast, but because a typical file system may include hundreds of thousands of files (Golden, 1998)) it may not be feasible or efficient to model and keep track of all files on the system.

Planning under partial observability requires that the planner be able to obtain information about the environment using sensing actions. Some actions may require that knowledge of particular fluents is established before executing the action (knowledge preconditions).

It then needs to be able to build plans which use those sensed values for control of action execution and for the values of subsequent action parameters. Example sensing actions from this domain and their properties are shown in Table 2.3.

The approach taken to sensing depends on the characteristics of the sensing action. For sensing actions which are cheap, quick and with no preconditions or side effects, such actions may be performed continuously and hence the fluent values sensed by these actions are effectively always known and may be referenced directly during plan execution. For sensing actions which are long running, costly, or have specific preconditions or side effects
Table 2.3: Example percepts

<table>
<thead>
<tr>
<th>Percept</th>
<th>Cost</th>
<th>Duration</th>
<th>Direct</th>
<th>Result type</th>
<th>Side effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check for file existence</td>
<td>None</td>
<td>1 sec</td>
<td>Y</td>
<td>Boolean</td>
<td>None</td>
</tr>
<tr>
<td>Check file size</td>
<td>None</td>
<td>1 sec</td>
<td>Y</td>
<td>Numeric</td>
<td>None</td>
</tr>
<tr>
<td>Check the depth of a message queue</td>
<td>None</td>
<td>1 sec</td>
<td>Y</td>
<td>Numeric</td>
<td>None</td>
</tr>
<tr>
<td>Check disk space</td>
<td>None</td>
<td>2 sec</td>
<td>Y</td>
<td>Numeric</td>
<td>None</td>
</tr>
<tr>
<td>Check how many processes are running</td>
<td>None</td>
<td>2 sec</td>
<td>Y</td>
<td>Numeric</td>
<td>None</td>
</tr>
<tr>
<td>Determine corrupted db record</td>
<td>100</td>
<td>2 hrs</td>
<td>Y</td>
<td>Numeric</td>
<td>DB offline</td>
</tr>
<tr>
<td>Determine if a disk has a hardware failure</td>
<td>None</td>
<td>5 min</td>
<td>Y</td>
<td>Boolean</td>
<td>Disk offline</td>
</tr>
<tr>
<td>Long running report indicates db or message reader is slow</td>
<td>10</td>
<td>15 min</td>
<td>N</td>
<td>Boolean</td>
<td>None</td>
</tr>
<tr>
<td>No heartbeat message and message queue up indicates settlement system down</td>
<td>1</td>
<td>5 sec</td>
<td>N</td>
<td>Boolean</td>
<td>None</td>
</tr>
<tr>
<td>Contact support to request status of an external system</td>
<td>5</td>
<td>20 min</td>
<td>Y</td>
<td>Enumerated</td>
<td>None</td>
</tr>
<tr>
<td>Determine name of oldest file in directory</td>
<td>None</td>
<td>1 sec</td>
<td>Y</td>
<td>String</td>
<td>None</td>
</tr>
<tr>
<td>Determine the process id of long running user query</td>
<td>5</td>
<td>3 min</td>
<td>Y</td>
<td>Numeric</td>
<td>None</td>
</tr>
</tbody>
</table>
it is not feasible to frequently and continuously execute such actions and instead the planner must explicitly build such sensing actions into the plan to be performed on an as-needed basis. This requires that the planner distinguishes the information available to execute the plan from the actual state of the world (which may be hidden). The planner must represent this information at runtime and must reason how to populate it from sensing actions. This includes achieving information for the purposes of both contingency control and for use as parameters to subsequent actions. As well as reasoning about knowledge acquisition, the planner must be able to reason about the effect of that knowledge on subsequent plan execution and what the effect of plan execution with invalid knowledge will be. The planner should also be able to represent and reason about contingent knowledge goals. For example there may be an externally defined goal such as 'in the event that the batch job A fails' then determine the cause of the error and communicate the cause to the support team, in the event that job A succeeds, just communicate the success of the job.

**Indirect sensing or diagnosis**

With direct sensing the value of a fluent may be expressed as a function of the output effects of the sensing action and hence the sensing action can be used to directly determine the value of the fluent (for example running a unix `ls` command to check for the existence of a file). For some fluents only indirect sensing actions exist where the output of the sensing action is some function of that fluent but the function is not information preserving and the fluent cannot be expressed as a function of the sensing action effect. In such cases, multiple sensing actions may be required to determine the value of that fluent from the results of multiple indirect sensing actions.

Where actions may be executed without their preconditions being verified (see section 2.2.7 on this) indirect sensing is often required to determine the failed precondition. (The lack of appropriate cheap, fast and direct sensing actions is often what necessitates the need for an action to be executed without precondition verification in the first place).
Existing approaches to sensing

The situation calculus may be extended with epistemic modal operators (Demolombe, Parra, & Pilar Pozos, 2000; Moore, 1985) which represent the agent’s beliefs about world fluents. These belief fluents may form part of action preconditions and postconditions. Golog has extensions which support sensing actions (H. Levesque, 1996). Knowledge is represented via an accessibility fluent $K(s', s)$ (Fagin, Halpern, Moses, & Vardi, 2003) which defines whether in the situation $s$ the agent believes it could be in situation $s'$. I.e. world $s'$ is epistemically compatible with world $s$ — $s'$ is equivalent with respect to an agent’s knowledge state in world $s$.

Knowledge is defined in the standard way where $\text{Know}(\phi, s)$ is an abbreviation for the formula $\forall s' [K(s', s) \supset \phi(s')]$. A sensed fluent function $SF(a, s)$ is introduced which provides the value of an observation made by action $a$. Knowledge preconditions and postconditions may be defined for actions. Since this approach only allows sensing fluents to be directly assigned from sensing actions, this approach does not support indirect sensing actions.

The event calculus (Forth & Shanahan, 2003) describes extensions to the event calculus to represent knowledge. A knowledge fluent $kw(f)$ is used to represent knowledge of whether the fluent $f$ is true or false. These knowledge fluents have their own axioms and inferences for reasoning about how actions affect knowledge fluents in terms of knowledge fluents either initiated or terminated by an action. The approach allows the inference of knowledge of unobservable fluents using knowledge versions of state constraint axioms.

Model based planners represent knowledge implicitly — representing state as a belief state over all fluents. Observation actions then update this belief state based on the results of the observation (which may be non deterministic). All planning is performed over these belief states.

Graphplan (D. S. Weld et al., 1998) has been extended to allow sensing actions. It uses a discriminator proposition $K\neg u : v$ to indicate that if the agent is in world $w_v$ then it knows it is not in world $w_u$ — i.e. that $w_u$ and $w_v$ are no longer accessible in the Kripke structure. Hence Graphplan’s knowledge goals are essentially to be able to discriminate between two possible worlds in order that the planner may perform the requisite contingent
PKS (R. Petrick & Bacchus, 2002) represents all actions completely in terms of their knowledge preconditions and effect on a set of 4 knowledge bases containing formulas in a modal logic of knowledge. $K_f$ which contains a set of formulas involving fluents in the world, $K_w$ is a database containing all fluent formulas $\phi$ for which the planner knows either the truth of $\phi$ or it knows the falsity of $\phi$, this database contains formulas upon which the planner can predicate conditional actions in the plan. $K_v$ is a database which contains functions whose value the planner will have access to at run time (although at plan time it may not know the specific value of the function). $K_x$ is a database which contains formulas defining knowledge of exactly one of a set of possible literals. The planner is able to perform temporal projection on the contents of these databases using the known updates which actions perform on the knowledge base. A goal $g$ is deemed to be achieved by a conditional plan when $K(g)$ eventually is inferred from the knowledge state on all possible executions of the plan. The planner performs both domain knowledge and consistency updates on the knowledge base following updates made by application of an action.

A belief set may be maintained (Halpern, 2005) which is the most plausible state of the external world which is consistent with the agent’s current knowledge. Plausibility may be defined using different approaches (simplest explanation, most probable etc). Depending on the form of the plausibility ordering and on whether the external state of the world is constant, the belief revision may or may not be monotonic. Usually the belief set is closed under inference. The limitation of this approach is that always establishing beliefs based on plausibility does not necessarily cause the agent to achieve an optimal conditional plan since the cost of establishing beliefs versus the impact of false beliefs (which varies from situation to situation) is not taken into account.

The Puccini planner (Golden, 1998) based on SADL does not use a modal logic representation and instead uses the concept of run time variables (which can take on specific values or a special “Unknown” value) to represent the agent’s knowledge. These variables are populated from the results of observations from sensing actions and may then be used for subsequent action parameters. Specific knowledge preconditions may be defined for actions.
The action representation language SADL (Golden & Weld, 1996), a derivative of ADL supports conditional, and observational effects. The effects of action are world state changes and world state reports, the latter assigning values to run-time variables. The use of a three-valued logic allows SADL to model uncertainty explicitly but PUCCINI does not support contingent planning. The contingent Cassandra planner (Pryor & Collins, 1996) uses decision conditions to represent the results of observations and to control plan execution.

Puccini does not represent contingencies (handling uncertainty is supported by interleaving execution with planning). The sensing model supports direct sensing actions only since the runtime fluents may only be directly assigned the results of observations and is not able to perform the contingent case-based reasoning needed to establish knowledge via indirect sensing.

The Cassandra planner performs contingent reasoning but assumes that all decision conditions may become known by a single sensing action so is unable to form a decision by using indirect sensing actions.

**Existing approaches to diagnosis**

The term diagnosis is usually used in connection with determining the cause of a problem or failed action. As such diagnosis of a failed action is only applicable in plans where actions with unverified preconditions are allowed (see subsection 2.2.7). The reason that an action may have unverified preconditions is usually because the sensing actions to verify those preconditions are too costly, long running or have undesirable side effects. Hence diagnosis often involves expensive or indirect sensing. The field of diagnosis has specialised and highly efficient algorithms for inferring causes of failure from the results of indirect tests. Different approaches to diagnosis have been identified in the literature: De Kleer’s model based approach (De Kleer & Kuriem, 2003; Forbus & De Kleer, 1993) is particularly suited for diagnosis of component failure (e.g. diagnosis of logic gate failure in a circuit using observations of outputs from the gates). It is capable of determining the minimal sets of the failed components which are consistent with a static set of observations. It can handle both boolean and numeric constraints between components. The approach can also determine efficient probing strategies for minimizing the cost of diagnosis based on the cost...
of measurements. It can also be extended to take into account the probabilities of each type of failure. Assumption based truth maintenance and constraint propagation techniques are used in these algorithms. The GDE algorithm implemented by de Kleer and Williams and follows the following phases.

- Behaviour prediction.
- Conflict detection.
- Diagnosis generation.
- Discrimination between diagnoses by additional measurement.

Struss (1997) gives a discussion of the first order logical foundations of model based diagnosis and outlines the advantages of such an approach versus a direct representation between fault and associated symptoms.

Autonomic approaches to problem diagnosis and correction (Sun & Weld, 1993; Littman, Nguyen, Hirsh, Fenson, & Howard, 2003; Symbium Inc, 2002; Janssen, 1989) utilize Bayesian models (Poole, 1997) to apply abductive reasoning and use value functions to take into account the cost of tests and perceptions when diagnosing problems. However these approaches are best suited to environments where the prior probabilities of each class of problem are known. In the case of a custom batch environment, the prior probabilities are generally not known (although if the same problems reoccur on a regular basis, it could be possible for the agent to learn these values.)

Littman et al. (2003) models cost of tests and remedial actions. States are modeled as set of possible classes or problem. A dynamic programming approach is used to provide accurate estimates of the expected cost to correction. Examples include disk diagnosis and network diagnosis and remediation. However remedial actions are assumed as singular actions so the approach would not work for domains requiring multiple corrective actions to achieve the remediation.

In Sun and Weld (1993) an integrated approach to planning and diagnosis is implemented where a diagnosis engine is called as a subroutine from within a planner. The diagnosis engine models probabilities and is able to generate efficient probing strategies by determining
the expected cost of each sensing and repair plan based on probabilities of abnormalities and costs of each sensing and repair action. UWL is a language used to represent the planning requirements and is able to model both actions that change the world and actions which sense values in the exterior world.

Baral, McIlraith and Son’s approach (Baral, McIlraith, & Son, 2000; McIlraith & Scherl, 2000) builds upon model based diagnosis and can perform diagnosis of malfunctioning components, but it is also capable of generating an explanation for observation narratives using narratives of exogenous actions.

Specialised diagnostic approaches have highly optimised algorithms capable of handling systems consisting of large numbers of components, but since they are not situated in a broader planning and sensing context they do not encode a concept of knowledge. Neither do these algorithms incorporate any model of time for the sensing or repair actions. Diagnostic mechanisms (Poole, 1997) should ideally be integrated with the planning and perception agent capacities. The agent ought to have the ability to plan with actions which have causal effects (traditional planning) and to plan with those actions which have only an observational effect (diagnosis). As discussed in Sun and Weld (1993) an integrated repair system should have both.

In order to integrate diagnosis with planning, the planner must formulate a knowledge goal to diagnose the cause of a failed action, perform the diagnosis and then use the resulting knowledge to repair the problem. There are different approaches to modeling knowledge — this is an extensive field in its own right (see Fagin et al. (2003) for an in depth analysis of knowledge representation). Planning to obtain knowledge with sensing actions which deliver indirect information about a fluent is more difficult than planning about actions which sense the fluent directly. For indirect sensing an action may not individually fulfill an information goal, but a combination of different sensing actions may be able to ascertain what the value of a fluent is.

As far as possible we would like the agent diagnostic abilities to be fully integrated with its planner rather than executed as an independent subroutine at runtime. Anticipation of unsuccessful actions should include sensing actions to determine the cause of failure and repair actions included in the plan which are invoked based off the results of these sensing
actions. By anticipating these actions at plan generation time the resource and duration side effects of the sensing actions may be taken into account when planning for goal achievement. For the batch job domain the typical problem size is small and is currently handled by the non specialized 'common sense' reasoning applied by the human support staff, so there is not necessarily a requirement for high efficiency algorithms. For the domain it is sufficient to support diagnosis over a small set of possible causes and to have that diagnosis arise out of the planner sensing.

2.2.5 A compact human readable plan representation

The plan representation should be easily readable by a human since the human operator needs to understand when a particular job will execute and help trouble shoot if during plan execution a situation arises which is not anticipated in the plan and the plan fails to deliver it goals. We would like a plan representation which is able to represent the contingent execution branches as compactly as possible for reasons of both efficiency of representation and to facilitate human readability.

State to action representations such as those used by model based planners such as MBP (Bertoli et al., 2001) (which represents a plan using a state transition relation), Schoppers (1987) (which uses a state+goal to action decision tree), or Markov decision planners such as mGPT (Bonet & Geffner, 2005) which represent the plan as a policy mapping states to actions are all compact representations of contingency since such representations can support the re-merging of execution branches. Figure 2.1 shows a contingent plan policy which involves the re-merging of execution paths. This plan handles two possible initial states s1 and s2 and achieves a goal state s4 by first achieving state s3 and then applying action C to s3 to achieve s4. State s3 is achieved by applying action A to s1 or action B to s2.
\[ \pi(s_1) = A \]
\[ \pi(s_2) = B \]
\[ \pi(s_3) = C \]

(2.1)

Note that although the plan involves two execution paths \((s_1 \implies s_3, s_3 \implies s_4)\) and \((s_2 \implies s_3, s_3 \implies s_4)\) which both pass through state \(s_3\) the policy represents this state only once. This representation therefore allows an exponential number of execution paths to be represented by a linear sized plan.

In a branching tree representation such as is used by SensoryGraphPlan (D. S. Weld et al., 1998), PKS (R. Petrick & Bacchus, 2002), each execution path corresponds to a leaf in the tree, so the plan size scales with the number of execution paths. Table 2.4 shows the branching tree procedural equivalent plan to 2.1.

<table>
<thead>
<tr>
<th>Table 2.4: Procedural plan equivalent of policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
</tr>
<tr>
<td>if ((s_1)) then</td>
</tr>
<tr>
<td>exec A</td>
</tr>
<tr>
<td>exec C</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>exec B</td>
</tr>
<tr>
<td>exec C</td>
</tr>
<tr>
<td>endif</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

2.2.6 Planning with exogenous events which are triggered by the occurrence of particular conditions.

The agent must be able to reason about exogenous events as not all changes in the computer environment are the result of agent actions. For example a file may be generated by an external system not under the control of the agent, or support teams may take actions
on the system. These events need to be modeled and reasoned about within the planning representation. Two forms of exogenous events must be handled — those triggered by conditions in the environment where those trigger conditions are brought about by the planner and those which are triggered by conditions which are not influenceable by the agent. To model exogenous events which could occur at any time as deterministic triggered events the event trigger conditions must be modeled as unknown values and the event duration as indeterminate.

Of the prior plan representations and planners Golog and the event calculus have language extensions supporting exogenous events, and MDP’s can model exogenous events by using uncertainty in transitions.

2.2.7 Execution of actions whose preconditions have not been verified as holding

If the sensing actions to verify all preconditions of an action either do not exist or are too costly or slow, the planner may need to execute an action without verifying prior to execution that all the preconditions for successful execution of the action are met Morgenstem (1987). Some preconditions may be denoted as secondary (not required to be confirmed before execution) and appropriate corrective actions taken later if these secondary preconditions are subsequently found not to have been true (this approach is employed by Golden (1998) in his 'leap before you look' mechanism).

For a plan which executes an action with unverified preconditions, the planner will need to construct a plan to handle contingencies where the preconditions are met and the action is successful and those contingencies where the preconditions are not met and the action is not successful.

For example a report batch job may require access to an application server applicationServer and a separate database server dbServer. The report will only be generated successfully if both applicationServer.state = Good and dbServer.state = Good.

If there are no fast and cheap sensing actions which determine the internal state of the application server and database the planner should not include explicit checks on these preconditions before executing the report job. Under nominal conditions where the application
server and database state are good, the report is generated and the job completes with a zero exit status. However if the application server or database state is not good then the report will not be generated and the job exit status will be non zero.

The planner will also need to include in the plan appropriate contingency control dependent on the exit status of the job. In the first contingency it will not need to take any action beyond running the report batch job. In the other contingency it will need to create a knowledge goal to determine which precondition was not satisfied using the appropriate sensing actions, correct the precondition and then re-run the report job. (The diagnosis of the root cause of the failure can then be viewed as a form of belief revision of the previously assumed nominal values of these preconditions.)

State based planners such as Markov decision process and model based planners are able to support unverified preconditions since a state which entails an action may be an encoding of the planner belief state which if established on a plausibility approach does not require strict knowledge of the action preconditions.

Languages and planners which define knowledge preconditions for actions (such as Golog, PKS) are not able to execute actions unless knowledge of those preconditions is contained in the knowledge base. Puccini achieves the ability to execute actions without ensuring all knowledge preconditions using a specific mechanism which allows the execution of actions with retrospective confirmation of whether or not the secondary preconditions for the action were actually met.

Depending on the available sensing actions and their costs, the planner can determine the cause of a failed job in different ways.

- It can perform an explicit sensing action on every precondition for the job (possibly with high cost and duration).
- It can infer the failed job precondition from existing observations and domain constraints. E.g. given the observation:

\[ tradeLoader.performance = Good \]
and the following domain constraint:

\[ \text{database.performance} = \text{Bad} \implies \text{tradeLoader.performance} = \text{Bad} \]

the agent should infer:

\[ \text{database.performance} \neq \text{Bad} \]

And from the observation:

\[ \text{report.status} = \text{Fail} \]

and the domain constraint:

\[ \text{report.status} = \text{Fail} \iff \text{database.performance} = \text{Bad} \lor \text{applicationServer.state} = \text{Bad} \]

the agent can infer that the root cause of the report failure is:

\[ \text{applicationServer.state} = \text{Bad} \]

• It may run external actions to bring about all the preconditions without bothering to
diagnose which one is actually false — for example the agent could assume that the
cause was the database server and restart the database server (this is similar to the
standard benchmark bomb dunking planning problem (R. Petrick & Bacchus, 2002)
where the agent cannot detect which package contains the bomb, but it can still ensure
that the bomb is diffused by dunking all packages.)
Under certain scenarios it may be more efficient for the agent to act under uncertainty in the external world rather than perform diagnosis and then act with certainty. For example, in the following situation. Job A (which takes 5 minutes to run) fails and from the precondition of the jobs, the planner knows that the cause must be either a database problem or a disk space problem or an internal problem with the program. The following sensing actions are available:

- Disk space can be sensed by running a disk check command (5 second runtime).
- Database state can be diagnosed with a database check command (1 hour runtime).
- Internal problems with the program can be diagnosed and fixed by contacting the support team (2 hour runtime).
- Disk space may be cleared with clear space command (2 minute runtime).
- A database restart command sets the state of the database server back to a good state (10 minute runtime).

Forcing diagnosis of the cause of failure would require the agent to run the disk space check command, the database check command and the internal program diagnosis with a total plan execution time of 2 hours, 5 minutes (assuming the sensing actions can be run concurrently). If there is a deadline < 30 minutes by which the job run has to successfully complete this plan would never meet the deadline. An alternative plan would be to clear the disk space, restart the database server and re-run the job (a total of 15 minutes, 5 seconds). If the job still fails then contact the support team to diagnose and fix the internal program error (2 hrs 15minutes, 5 seconds total runtime). With this alternative plan in the case that the root cause is disk space or database error the job will run successfully by the deadline.

2.2.8 Summary of Features Versus Planners

Table 2.5 provide a summary review of each of the planning features against the discussed planners, showing that no single planner incorporates all of these features — the key issue being that planners which handle durative actions do not handle sensing and vice versa.
Table 2.5: Existing Planner Feature Support

<table>
<thead>
<tr>
<th>Planner Feature</th>
<th>SIPE</th>
<th>PRODIGY</th>
<th>Puc-cini</th>
<th>PKS</th>
<th>Golog</th>
<th>Ab-demo</th>
<th>MBP</th>
<th>mGPT</th>
<th>SGra-philplan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action logic (if applicable)</td>
<td>Act</td>
<td>PDDL</td>
<td>SADL</td>
<td>Sit. calc.</td>
<td>Sit. calc.</td>
<td>Event calc.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numeric inequalities</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Concurrent actions</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Durative actions</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Execution Monitoring</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Goals with time windows</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Human readable compact plan rep-</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>resentation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct sensing</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Indirect sensing</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Automatic sensing</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Actions with unverified precondi-</td>
<td>N/A</td>
<td>N/A</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>tions.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triggered and exogenous events.</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
2.3 Determining the domain requirements

The aim of the problem case study was to frame the key requirements of the domain in terms of AI planning and diagnostic abilities in order to determine whether existing representations from the literature would support these features or whether a new representation would be needed. From this study it is shown that even this apparently straight forward problem domain requires a broad set of integrated AI planning features which is consistent with the findings of Golden (1997) and Goldman and Baral (1998) in their assessments of the needs of their softbots.

2.3.1 Batch job scheduling

As discussed in the introduction, batch job scheduling is a costly real-world problem whose characteristics (designed and deterministic environment, software based actions, controlled environment) make it a good candidate for automation using AI based planning and scheduling techniques.

2.3.2 Existing automation approaches

Commercial automated schedulers exist for the timed launch of batch jobs and basic monitoring of their return status (ComputerAssociates, 2002; UC4Corp, 2002). The scheduler runs a job (by executing the defined command for that job) when the defined start conditions for the job become true. The schedulers have an event processor which is able to monitor these conditions and execute the job as soon as the conditions become true. The start conditions can include the execution status of other jobs, the existence of specific files, or time conditions according to the needs of the job. For example a job runReport which has to run after the end of the business day, requires an input file (which is received from an external system) and which requires access to a database might have the following start conditions defined:
Such start conditions must be defined by the support staff as part of the job schedules they create according to the known behaviours and dependencies of the jobs.

Rudimentary retry logic is provided by these tools whereby a batch job may be re-run a number of times up to some predefined maximum number of retries. Some of the schedulers also have a facility which allows the user to view upcoming job runs in order to help the user to detect schedule conflicts. However the problem of diagnosis and error correction for failed jobs remains a manual process, requiring a large number of well trained support staff, who must be on call while these jobs run.

Previous approaches to the automation of batch job control (Ennis, 1986), have utilized a pattern based rule approach to error situation identification. The disadvantage of this approach is that rules must be hand-coded by a skilled operator for each computer environment and for each possible situation which needs to be handed. Ideally the agent should have the same flexibility of a human operator, who given just a high level qualitative description (e.g. its priority, data inputs and outputs etc) of the suite of computer batch job is able to support the running of those jobs.

2.3.3 Approach for determining the planning and diagnostic features needed to create an automated agent for this domain

The specific forms of reasoning required by an agent were determined by both considering the general characteristics of a computer environment (discussed in the previous section) and by performing a detailed case study of a real-world production batch job environment.
Figure 2.1: Case Study System
2.3.4 Case Study

The purpose of the case study was to frame the requirements of an agent for this domain in terms of planning and diagnostic AI techniques and to determine whether existing representations could handle the requirements of this domain. The case study was performed on a real-world trade processing production batch job environment.

System architecture of the batch job environment

The case study batch job environment consisting of the following entities:

- **Database Server** — a production database server which is available at all times except when it is undergoing weekly maintenance (a weekly database maintenance batch job which starts at 9 a.m. on Saturday with a typical duration of 2 hours). The server holds a database table called *Trades*. The database services queries and updates requested by the application server.

- **ApplicationServer** which accesses the *Trade* table in the database server. This is a daemon process which runs daily from 7 p.m. — 6:30 p.m. Sunday — Friday. The application server receives requests from client processes and makes appropriate calls to the database server to service those requests.

- Various files such as *tradeReport, tradeInput*.

- Message queues which are used to provide communications with external systems.

- **DailyAccounting** is a daily client batch job which runs at the end of day Monday through Friday at 6:30 p.m. It calculates total trade volumes for each day.

- **TradeLoader** is a daily client batch job which runs at the beginning of the day, 9 a.m. Monday through Friday. The job reads trade data from a file (trade input feed file) which is created by an external system and loads it into the *Trade* table via the application server.

- **TradeReporter** is a daily client batch job which runs at the end of the day; 6 p.m. Monday through Friday. The job reads trade data for that day from the *Trade* table...
and creates an output file (trade report) which it ftps to an external system.

- *TradeSettlement* to bank is a daemon process which runs daily 7:15 a.m. — 6:15 p.m. Sunday — Friday. It extracts trades ready for settlement from the database (via the application server) and sends payment messages to an external settlement system via a message queue (outbound message queue).

- *TradeSettlementStatus* is a daemon process which runs daily 7:15 a.m. — 6:15 p.m. Sunday — Friday. The daemon reads settlement status messages from the inbound message queue (which are sent back by the external settlement system) and updates the status in the *Trade* table accordingly.

The architecture of this environment is typical in structure to many commercial computer batch job environments.

### 2.3.5 Methodology

For the case study I documented a set of 20 observed scenarios involving interactions of the human operators with the computer environment covering a period of 6 months of the system operation. The observations were described in a standard format to allow the extraction of the essential forms of reasoning and actions performed by the human operator in order to handle the situation. The observed scenarios had the following general form:

- A schedule exists to execute a set of batch jobs at predetermined times and with defined start conditions.

- During execution of the schedule, one or more failures occurred (e.g. a job did not complete successfully, or it did not produce the desired results). The scheduler notifies the operator that one of the job fails, or a downstream system notifies that something is wrong with one of the job outputs (observed problem sensing).

- The human operator performs relevant tests and from the observations determines the root cause of why the job/process failed in its execution (diagnosis, plan for sensing).
• The human operator performs new actions which are designed to rectify the identified problem and achieve the original goals of the plan (plan for corrective actions).

Table 2.6 shows some of the key example scenarios (for the complete set of scenarios, please see the appendices):

Table 2.6: Key example scenarios from case study

<table>
<thead>
<tr>
<th>Title</th>
<th>Observed problem</th>
<th>Root Cause</th>
<th>Corrective action</th>
<th>Required capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource requirements</td>
<td>Data input load fails</td>
<td>Lack of disk space</td>
<td>Clear up disk space. Rerun input load script</td>
<td>Numerical condition effects</td>
</tr>
<tr>
<td>Event monitoring</td>
<td>Send payments file, acknowledgement not received in required response time</td>
<td>External payments system down</td>
<td>Notify external team</td>
<td>Event monitoring, Durative events, Exogenous event.</td>
</tr>
<tr>
<td>Job overrun</td>
<td>DB Dump process does not complete by expected end time</td>
<td>DB internal error</td>
<td>Alert DB team.</td>
<td>Temporal constraints, exogenous event</td>
</tr>
<tr>
<td>Concurrent actions</td>
<td>run 3 jobs to generate 3 required report types. Balance report fails</td>
<td>DB block</td>
<td>Unblock DB, rerun balance report then generate overall marker file</td>
<td>Concurrent action monitoring, Direct sensing, Unverified preconditions</td>
</tr>
<tr>
<td>DB error diagnosis</td>
<td>DB start job fails</td>
<td>DB index error</td>
<td>Run multiple db diagnostics to determine error. Run DB repair job for appropriate error, restart DB</td>
<td>Knowledge goal using indirect sense actions.</td>
</tr>
</tbody>
</table>
(based on the required schedule information) for each of the jobs and processes and execute and monitor the plan. A simplified form of the example system with a database, message queues, application server etc was set up to run as a test bed. Some of the problem scenarios were re-created in the test system by re-creating the root cause and the agent’s responses (or lack of responses!) to the problem were observed.

This prototype provided a basic proof of concept for the agent and enabled identification of the required abilities by noting the shortcomings of the agent on some of the scenarios. The agent was implemented in CLIPS (Giarratano, 2007; Jackson, 1999) which is a production rule system which has object oriented abilities. It was implemented as a classical non contingent partial order planner with threat resolution (see Russell and Norvig (1997); D. Weld (1999) for definitions of these planning techniques). By running the agent and observing limitations and problems with the planner’s behavior, I was able to identify the additional required abilities of the agent. Where possible extra abilities were implemented in the agent and the agent re-run on the scenarios. During this iterative phase some of the required features were added to the prototype. Although some of the specific requirements were handled by the prototype agent it had fundamental shortcomings — in its inability to handle contingent planning and knowledge representation.

The abilities of the prototype planner were:

- Handling of inequalities for numeric fluents — Pre and post conditions of the form \( x > 20 \) were introduced (where \( x \) is any numeric fluent)

- Simple automatic perception cycle — All available percepts are performed continuously during the execution cycle of the planner.

- Heuristics — cheaper actions are chosen as candidates in the plan search before considering more costly actions. Conditions with the fewest number of means of support are fulfilled first in order to reduce backtracking. Causal support is first tried from events which are closer in time to the event needing support — in order to minimise the possibility of threats.
Expressing the requirements

To consider the requirements in the context of the existing body of work on artificial intelligence techniques, and to determine what was possible with existing planning techniques and what aspects required new representations the following representational questions were posed and answered for the environment:

- How are the various components in the computer system architecture modeled and how does the structure of the system define the system’s behaviour?
- How are states represented and what forms of action preconditions and postcondition need to be represented?
- What kind of temporal constraints and goals need to be planned for. How are action durations defined and how must action execution be performed?
- Do hierarchical representations of actions need to be used?
- How does non determinism need to be modeled?
- Are all/some fluents observable? What forms of knowledge does the planner need to represent and plan for. What forms of sensing action are available to the planner. How does the planner formulate knowledge goals?
- How does the planner store the results of observations and how does the gained knowledge control the execution of the contingent plan?
- How are contingent plans represented. How does the planner detect when non nominal situations arise?
- Does the planner need to be able to execute actions without verifying that all preconditions for successful execution have been met?
- What forms of exogenous events need to be represented?

By answering each of these questions for the general properties of computer environments and for each of the specific scenarios documented during the case study (see appendix for details) the requirements listed in Table 2.2 were identified.
Chapter 3
NEW INTEGRATED PLANNING AND DIAGNOSTIC REPRESENTATION

This chapter presents a new plan representation ‘JobPlan’ which is able to support the
needed features in an integrated way. The representation design decisions are discussed and
an description is provided of how the representation supports each of the required features.

3.1 Requirements of the Representation

The plan representation must support the requirements identified in the previous chapter.
The next section describes the new plan representation which was formulated to address
these requirements.

3.2 Formulation of New Representation

The JobPlan representation uses a simple form of program description to describe the plan.
The explicit ordering sequence of actions is not stipulated as in a standard procedural
language, instead the execution of each action is triggered by the occurrence of a defined
set of start conditions for that action. I.e. the plan is defined as a policy — a set of state
to action mappings where the states are defined in terms of conditions which hold in that
state. The execution semantics of the plan is similar to that of coloured petri nets (Jensen,
1995) where the planner actions occur as soon as the action is enabled (i.e. its tokens
become true). Once an action is executed then the effects of that action eventually take
place upon both the plan state and the external world state. In this domain, the effect
of an action or event does not occur immediately after the action is triggered. The effects
occur some duration after the action or event is triggered where the duration may be non
deterministic. Since actions have a duration, concurrent actions and events are allowed and
the ordering imposed on the action execution is therefore a partial order. Plan generation must take into account these durations. Although the plan explicitly represents only the planner actions, during plan deliberation the state of both the plan and the external world and exogenous events is taken into account.

The representation combines aspects of the following forms of plan representation:

- The “plan as program” approach of Stephan and Biundo (1993); H. J. Levesque et al. (1997), (Nau et al., 2004, Ch. 12). In this approach the plan is described using a plan programming language and the execution of the plan corresponds to execution of the program. In the case of the Golog planning language the effects of an action or a plan are defined using the situation calculus with predicates such as the action execution command \textit{Exec} and test operators such as “?” which control contingent plan execution. Golog is a relatively high level procedural language which allows execution control according to sequential, conditional, and loop constructs. Such languages require a relatively sophisticated interpreter to convert these programs into an execution of actions. Although expressive languages allow a higher level (and more compact) representation of plans, plan/program construction becomes a difficult task requiring extensions to first order logic.

- Model based planning (Nau et al., 2004, Ch. 17) uses a finite state machine representation which defines a non deterministic transition relation upon these states. Finding a plan which meets a goal consists of finding a plan which under this transition relation is guaranteed to eventually evolve into a given goal state.

- POMDP (Cassandra, Kaelbling, & Littman, 1994) plan representations where a plan is a policy which maps belief states of the planner to actions. The new JobPlan representation follows a similar approach with actions triggered by the occurrence of certain conditions.

- Partial order based graph representations of agent actions which define a partial order on the actions e.g. Golden (1997). Actions are ordered after other actions that furnish their required preconditions. For contingent planning, actions may be labelled to determine under which contingencies, they are executed (Pryor & Collins, 1996).
3.3 New Plan Representation

This representation draws upon the job schedule used by commercial job schedulers — where agent actions are executed as soon their associated start conditions become true. The representation’s viability as a runtime job execution and monitoring model which handles triggered events and durative actions has been demonstrated via its use in commercial schedulers. Additionally its readability and interpretation by human operators has been attested to. This simple ‘programming/workflow’ representation also obeys a simple formally definable dynamics and hence can be directly reasoned about for plan synthesis and validation using an appropriate action logic.

The plan is described as a set of action job and plan variables. The job definitions define which actions are executed by the plan and the conditions under which each action is executed. Each action job definition consists of the following attributes:

- The action command (a constant for a given job) — which provides the external action script which this job will execute — the external effects of executing the job will be those of the referenced script.

- A job status which shows the current execution status of this job: *Initialised* (job has not been started yet) / *Executing* (Job action is executing) / *Completed* (job action completed). This status forms part of the plan state and is implicitly set to *Initialised* in the initial state of the plan (hence this does not have to be explicitly defined in the job definition).

- A description of optional start conditions which define under which circumstances the job is executed. These define the trigger conditions for the job execution event.

The syntax for description of a JobPlan definition is shown in Table 3.1

An example plan in this syntax is shown in Table 3.2. This plan first executes the *getDate.sh* script which determines the current state and assigns the value into the plan variable *i_todays.date*. The script *runReport.sh* is then run with *i_todays.date* as a script parameter to generate the report for the date specified in the parameter. When *runReport.sh*
Table 3.1: JobPlan Syntax

\[
\begin{align*}
\langle \text{plan} \rangle & ::= ( \langle \text{jobDefn} \rangle^* \rightarrow \langle \text{varDefn} \rangle )^* \\
\langle \text{jobDefn} \rangle & ::= ("\text{name} : \langle \text{identifier} \rangle", \text{command : } \langle \text{command} \rangle, \text{startConditions : } \langle \text{startCondition} \rangle^* ) \\
\langle \text{varDefn} \rangle & ::= ("\text{name} : \langle \text{varName} \rangle", \text{value : } \langle \text{value} \rangle) \\
\langle \text{varName} \rangle & ::= \text{"i"} \langle \text{fluentName} \rangle \\
\langle \text{fluentName} \rangle & ::= \langle \text{identifier} \rangle \text{."} \langle \text{identifier} \rangle \\
\langle \text{command} \rangle & ::= \langle \text{externalCommand} \rangle | \langle \text{internalCommand} \rangle \\
\langle \text{externalCommand} \rangle & ::= [\langle \text{varName} \rangle =] \langle \text{identifier} \rangle [\langle \text{varName} \rangle] \\
\langle \text{internalCommand} \rangle & ::= \langle \text{varName} \rangle = [\langle \text{varName} \rangle] \\
\langle \text{operator} \rangle & ::= \text{" > "} | \text{" < "} | \text{" = "} | \text{" \# "} \\
\langle \text{constant} \rangle & ::= \langle \text{string} \rangle | \langle \text{number} \rangle \\
\langle \text{identifier} \rangle & ::= \text{\langle letter \rangle} | \langle \text{identifier} \rangle \text{\langle letter \rangle} | \langle \text{identifier} \rangle \text{\langle digit \rangle} \\
\langle \text{number} \rangle & ::= \langle \text{digit} \rangle^* \\
\langle \text{externalCommand} \rangle & ::= [\langle \text{varName} \rangle =] \langle \text{identifier} \rangle [\langle \text{varName} \rangle] \\
\langle \text{internalCommand} \rangle & ::= \langle \text{varName} \rangle = [\langle \text{varName} \rangle]
\end{align*}
\]

has completed the script *ftp.pl* is run with *i_todays.date* as a script parameter which ftp’s the report for the specified date to a remote server.

**Commands**

A command is either an external operating system command line or a description of an internal planner variable assignment. An external operation system command requires the name of the script to be run and the optional specification of a planner variable as command argument and the optional capture of the script return value into a planner variable (in the example above *job_getdate* which runs the script *getDate.sh* and assigns its output to the plan variable *i_todays.date*). An internal command consists of the assignment of a value (either another planner variable or a constant) to a planner variable. The command is run at the point the job status becomes *Executing*. During planner deliberation for each command there are corresponding event definitions used by the action logic to determine the effects of running the command. These definitions define both the immediate effects when the command starts and the effects once the command has completed under both nominal and non nominal conditions. A minimum of two event definitions are required for any command or four event definitions if non nominal execution is possible. The event
definitions include the effect on both planner variables and world fluents.

**Plan variables**

A _plan variable_ (whose identifier is prefixed with _i_ to distinguish it from external world fluent identifiers) defines a plan fluent and its initial value. These are a form of runtime variable (Golden, 1998). Such variables can be used for representing the value of fluents in the world. E.g. in the example above _i.todays.date_ is used to represent the current date and is set from the return value of a scripts which senses the current date value. Plan variables may be set either from the results of an external command script or internal planner command or by assigning a constant value or a simple boolean expression involving other plan variables. The value of a plan variable usually changes during execution of the plan. The value of these variables may subsequently be used to control execution of an action job (by including a condition on the plan variable as a start condition of the job), or as input parameters for an action command (by referencing the plan variable value in the command parameters). For example in order to only run a command to restart the message channel if the channel is down the job would include a start condition that _i.channel.up = False_ The plan variable is reasoned about as an _internal_ fluent within the plan. (See the following section on knowledge representation on how these plan variables /
internal variables are used for knowledge representation and sensing).

Start conditions

A job’s start conditions are a set of formulas describing conditions involving either a plan variable, job status or the value of a world fluent whose value is continuously sensed and accessible to the planner at runtime (such fluents are called automatic fluents since their values are automatically known by the agent and do not have to be explicitly sensed). Sequential orderings between actions are stipulated by including the completed status of other jobs as start condition to the successor jobs. Conditional plan execution is achieved by including conditions on these plan variables in the start conditions for contingent action jobs. Hence depending on whether a job’s start conditions ever occur during plan execution an action job in the plan may or may not be executed. The plan representation therefore combines the ability to stipulate action sequence representation with event based triggering of actions. In the example above the job job_report runs as soon as the fluent inputFile.exists becomes True.

3.3.1 Plan execution

Execution of the jobPlan is very simple. During plan execution the executor constantly monitors all sets of job start conditions for those jobs in Initialisedstate. (This is why start conditions must be formulas involving either plan variables or world fluent values to which the scheduler has direct and continuous access). As soon as all the defined start conditions for a job become true the executor immediately initiates execution of either the batch job via the operating system command line using the specified parameters values, or (for internal commands) initiates the internal planner command and sets the job status to Executing. When the command has finished executing (which the plan executor is immediately able to detect when the command operating system process completes) the planner moves the job status to Completed. The time of command completion $te = tt$ (time of command initiation) + $d$ (command duration).
3.3.2 Reasoning about the plan execution dynamics

In order to determine how the world evolves under the plan, the state of the plan, the state of the world and the dynamics of both the plan and the world must be taken into consideration. All change is either due to planner actions or exogenous events. The planner’s execution of a batch job consists of the run of a program. The internal state of the batch job program consists of the combined state of each of its variables, data buffers, output files etc. This state evolves continuously over time with a change in program state occurring at every time step during the run of the program. In order to abstract away from these continuous changes for planning purposes a simplification is made in which only the high level fluents of the batch job are represented and the changes represented as those changes which take place immediately upon execution of the batch job and those other changes which have occurred by the time the job completes. The high level characteristics include the overall job status (Executing or Completed), resource usage, key inputs and outputs (files, messages etc). In the action logic used to reason about the evolution of the plan, all events have a single set of effects all of which take place at the same time. Since the run of a batch job involves both effects which occur as soon as the job is started and effects which occur once the job is completed, two events are needed to describe all the effects of running a job — the action start event and the action execution event.

Action Start Event — This event describes the initiation of the job command and its trigger conditions are defined as the start conditions for the job. As soon as all of the conditions defined in a job’s start conditions become true and the current job state is Initialised the job state changes to Executing. If the job command is an external batch job program the effects of this event include those immediate effects of starting the batch job (such as increases in resource usage, move of input files etc). The effects of this event are considered as immediate — i.e. the effects of the event are assumed to take place at the next time point from when the event is triggered.

Action Execution Event — This is the event which describes the changes brought about by the full completion of the batch job. The event is triggered when the job status is Executing. If the command is an internal command (such as a planner variable assignment)
that command is performed and any update to the specified planner variable takes place (for example the effect of a planner variable assignment action might be $i_x.errorCode = 1$). If the command is a batch job the event describes the changes which take place between the job first being initiated and when it completes. This includes effects such as generation of output files, release of resources held by the batch job execution etc and assignment of any returned results from the command to the specified plan variable. For both external and internal commands the effects of this event also include the change that the job status is set to *Completed*. Batch job executions have an extended duration $d$ which can vary between the different runs of the job and is dependent on factors which are not modeled such as the specifics of the data inputs of the program.

*Exogenous events* — these are events which are not under direct control of the planner. Such events are therefore not represented as actions in the plan itself (since they are not directly actioned by the plan), but they must still be reasoned about during plan construction in terms of their potential occurrence (which may threaten the desired effects of the plan) or because the planner must interact with such exogenous events in order to meet the plan goals. Exogenous events are also represented as triggered events. Exogenous events which can occur at any point in time are represented as triggered events with unconstrained duration and with trigger conditions which are not sensable by the planner. With such events the planner has no means to determine whether or when such events occur — which means that whatever plans it constructs must work even if exogenous events can occur at any point in time during execution. For exogenous events that the planner does have the ability to influence, the exogenous event’s trigger conditions involve fluents which the planner can affect. Such exogenous events may be triggerable by the planner as needed in order to achieve the goal.

In the action logic the frame assumption is made for all events. All fluents in a state have the same value after any event — except for those effect conditions explicitly defined for the event. During plan construction, the planner creates new instances of action jobs and associated plan variables as needed. All jobs added into the plan initially have a status of *Initialised*. Since there is no action to set the status of a job back to *Initialised* each job may be triggered a maximum of one time, so the number of actions executed by a plan is bounded
by the number of jobs defined in the plan. Internal fluents are not subject to exogenous events and may only be changed by agent internal actions or sensing actions. In terms of correspondence to procedural programming languages, internal value fluents correspond to program variables, internal actions correspond to inbuilt programming operations and action jobs correspond to program statements. External actions correspond to input/output operations which communicate with the external world outside of the program.

The description of actions and events in terms of in the action logic used for generating and validating plans is described in Chapter 4.

In terms of access to the value of the time fluent it is assumed the agent will always have continuous access to the time fluent (hence time based conditions may be included as the start condition for action jobs). In terms of the time conditions which may happen to occur in a given trigger or effect state — those time conditions may vary by contingency — as may the value of any fluent conditions.

As described the representation and action logic is capable of supporting time based reasoning, however the current implementation of the planner does not implement temporal reasoning and has not been assessed on example problems using goals or event triggers or event effects which include time conditions. Extension of the planner to include temporal reasoning remains as future work.

3.3.3 Describing the plan dynamics using state and actions descriptions

State descriptions

During plan construction the planner must reason about the state of the plan and the world and how it evolves over time. Since contingent planning is required the planner must reason about the evolution of multiple plan+world states. In order to facilitate efficient reasoning about multiple states, the state description used is a partial description which specifies the values of only some of the state fluents. This partial description allows set of states to be succinctly described and reasoned about. The fluent values are specified by formula giving the set of conditions which hold in that state.
Event descriptions

As described above, events occur as soon as their trigger conditions become true and once an event occurs, the effects of that event are guaranteed to occur at some later point in time (determined by the event duration which can vary). An event can therefore be described by a set of trigger conditions and a set of effect conditions and an (optional) lower and upper bound on the event duration. Plan jobs must be generated such that when the plan state evolves over time the actions are executed within the necessary order to achieve the goals for all possible event durations. An action is described by a set of two or more events. An action which has no conditional effects is defined by two events, the start event for the action and the action execution event. The start event trigger conditions always include a condition that the job status is initialised and may contain other trigger conditions as defined by the start conditions for that job. The start event effect conditions includes the condition that the job status is *Executing* and may also include external effects such as changes to the value of resources which may occur upon start of the job. The action execution event effect conditions includes the condition that the job status is *Completed* and includes all other external effects of that job. For actions which have conditional effects, multiple action execution events may be defined, each of which has different trigger conditions which encode the status of secondary preconditions of the action. Each of these different action execution events has different effect conditions — describing the outcome of the same action under different circumstances. Typically an action may have two action execution events — a successful action execution event describing the successful execution of the job. This event will contain trigger conditions which include all preconditions which ensure successful execution of the job and will contain effect conditions which correspond to successful outcome of the job. The failure action execution event describes the unsuccessful execution of the job. This event will contain trigger conditions which preclude successful execution of the job and will contain effect conditions which correspond to an unsuccessful outcome of the job.
3.3.4 Plan generation

The semantics of the JobPlan representation is grounded in the same way as a description of the external world is defined — in terms of the value of fluents which describe plan state and the dynamics defining the evolution of the plan state. In terms of different execution paths the possible trajectories of the plan execution are defined implicitly in terms of the dynamics of the combined plan and world system. In this approach a clear distinction is made between the plan workflow representation (a set of action jobs, their start conditions and plan variables) and the planner deliberations which use an action logic to reason about the evolution of the plan and world. This is in contrast to most partial order planners which employ an overloaded representation where some of the plan components refer to items which control plan execution (plan actions, orderings), some are used solely for inference purposes (causal links, protection) and some may be used for both (preconditions — which are used during plan construction, but which may also be used for monitoring during plan execution).

3.3.5 Example sequential plan

The plan:

(name: job1,
   command: "runReport"
   startConditions:)

(name: job2,
   command: "ftp report"
   start condition: job1.status=Completed)

consists of two actions — job1 generates the report, job2 ftps the report to a remote location. The procedural pseudo-code form for such a plan would be as follows:

begin
   exec “runReport”
   exec “ftp report”
end
This plan is a classical plan with no contingencies and no sensing. The plan stipulates that the ftp is only performed after the local report has been generated by including the completion status of report job1 in the start conditions for the ftp job. Figures 3.2 — 3.5 illustrate the evolution of the state of the plan and world during execution of a simple plan which produces a report file on a remote location disk. See Figure 3.1 for the diagram convention. In the initial state the status of both jobs is *Initialised*. Neither local or remote report file exists. In Figure 3.3 job1 execution is triggered (it has no start conditions) and the job state changes to *Executing*. In Figure 3.4 job1 completes and the effects of the action on the outside world take effect (the local report file is generated). In Figure 3.5 job2 execution is triggered since its start conditions are now true, the job state changes to *Executing*. In Figure 3.6 job2 completes and the effects of the action on the outside work take effect (the remote report file is generated).

### 3.3.6 Example plan with external event monitoring

In this plan:

```
(name: job1,
   command: "runReport"
   startConditions: inputFile.exists=True)
```

the generation of a report requires the existence of an input file *inputFile*. Running the report job if *inputFile* is not available will not successfully generate a report. *inputFile* is generated by an exogenous event at some unspecified time. The procedural pseudo-code form for such a plan would be as follows:

```
begin
   while not inputFile.exists
   endwhile
   exec "runReport"
end
```

Figures 3.7 — 3.10 illustrate the evolution of the state of the plan and world during execution of a simple plan which produces a local report file. The plan consists of a single
Figure 3.1: Diagrammatic convention
Figure 3.2: Sequential Plan — initial state

Figure 3.3: Sequential Plan — executing 1st action
Figure 3.4: Sequential Plan — 1st action completed

Figure 3.5: Sequential Plan — 2nd action executing
action — job1 generates the report. The plan ensures that the report job is only run after the *inputFile* is available by including the existence of the file in the start conditions for the report job. (The file existence is an automatic fluent to which the planner has continuous and immediate access to and it therefore possible to reference this as a start condition).

In Figure 3.7 in the initial state the status of job1 is *Initialised*. Neither the *inputFile* nor report exist. In Figure 3.8 the exogenous event which generates the input file occurs and has the effect that *inputFile.exists* = *True*. In Figure 3.9 execution of job1 is triggered since all start conditions for the job are now true. In Figure 3.10 job1 completes and the effects of the action on the outside work take effect (the report file is generated).

### 3.3.7 Example plan with actions with unverified preconditions

An example of a plan with unverified preconditions:

```
(name: job1,
  command: "dunk p1"
  startConditions:)
(name: job2,
  command: "dunk p2"
  startConditions:)
```

is from the benchmark *bomb in a toilet* problem (McDermott, 1987). In the example presented here there are two packages *p1* and *p2*, one of which contains a bomb. The bomb is disarmed by dunking the package which contains it into the toilet. In this case it is assumed the toilet can hold more than one package so package1 can be dunked at the same time as package2. In order to disarm the bomb the package being dunked must contain the bomb. However, this precondition cannot be verified since there are no sensing actions to determine whether a given package contains the bomb. However by dunking both packages the goal may be achieved nonetheless.

Figures 3.11 — 3.12 illustrate the execution of the plan which achieves the goal that the bomb is disarmed.

Figure 3.11 shows the initial plan state. The bomb is in package *p2*, neither dunk action has taken place and the bomb is not disarmed.
Figure 3.6: Sequential Plan — execution complete

Figure 3.7: Event monitoring plan — initial state
Figure 3.8: Event monitoring Plan — exogenous event generates input file

Figure 3.9: Event monitoring Plan — report generate job starts
Since there are no start conditions for either of the dunk jobs, both jobs are triggered immediately and run concurrently.

Figure 3.12 shows the plan state after both dunk actions have completed execution. Since p1 did not contain the package the dunk p1 action has no effect. Since p2 contains the bomb the dunk p2 action has the effect of disarming the bomb and the goal of \( \text{bomb.disarmed} = \text{true} \) is achieved.

Note the same plan would work if p1 contained the bomb.

### 3.4 Representation of Beliefs and Sensing

#### 3.4.1 Using plan variables to represent beliefs

Fluents which are directly sensable with little or no cost and which may be continuously sensed by the planner are termed *automatic* fluents. Such fluents may be directly referenced in the plan (e.g. a start condition for a job can involve an automatic fluent). For fluents which are not automatic and where the planner needs to provide the value of the fluent as a parameter to an action, or as a start condition for an action, the planner generalises (via the ability of contingent reasoning) the Puccini approach of *runtime variables* and uses a plan variable (whose values is described by a fluent internal to the plan) to represent its belief about the value of the fluent. These program variables constitute the planner’s knowledge about the values of fluents in the world. When these internal program variable values correspond to the values of fluents in the external world such internal fluents are considered as correct beliefs. Since plan variables are used for action control believing X can be considered as meaning “behave as if X is true”. In this sense the beliefs are defined contextually via their usage, not syntactically.

As touched upon in the description of the plan representation, plan variables (i.e. internal fluents) are used to represent the results of sensing actions and to represent beliefs about the outside world. These variables are part of the plan and during plan construction the state of these variables must be reasoned about as well as the state of the world.

This approach avoids a modal logic representation of belief since the beliefs are defined concretely as the state of these epistemic plan variables within the plan itself and are
Figure 3.10: Event monitoring Plan — report generate job completes

Figure 3.11: Bomb dunk — initial state.
reasoned about as such. This avoids the need for specialised modal logical inferences and allows sensing to be performed using the same reasoning as is used for reasoning about changes to the external world. The knowledge representation does not employ a syntactic representation — each belief about a fluent value or proposition in the external world is presented a separate plan variable. This means that the fluents or propositions about which the planner has beliefs must be defined at plan generation time with the values of these beliefs being populated at plan runtime according to the sensing actions specified in the plan. This is different from agents whose knowledge base consists of formula of known propositions and which have the property of consequential closure where the knowledge base contains all the logical consequences which follow from the knowledge base. For example if $\phi \iff \psi$ then if the knowledge base contains $\phi$ it must also contain $\psi$ and if the knowledge base contains $\neg\phi$ it must also contain $\neg\psi$. With the plan variable representation such consequences would need to be determined at plan generation time, not at run time. For example if the plan goals require that the values of both $\phi$ and $\psi$ be known the planner might create two plan variables $i_{\text{phi}}$ and $i_{\text{psi}}$, populate $i_{\text{phi}}$ from a sensing action for $\phi$ and then add an action which assigns the value of $i_{\text{phi}}$ to the value of $i_{\text{psi}}$.

3.4.2 Belief representation and establishment

A belief is considered to be an internal plan variable whose context and usage within the plan dictates that it refers to the value of a fluent in the external world. A belief is accurate when the value the plan variable holds is the same as the value of the fluent in the external world. The representation does not enforce a strict requirement that beliefs are accurate in all worlds where the planner holds that belief (i.e. it allows for the planner to hold false beliefs). During plan construction for each plan variable a subgoal is set up to establish accurate beliefs - i.e. that plan variables correspond to the fluent it represents at the point the plan variable is accessed for either plan decision control, or as an action parameter. If the planner is able to achieve this subgoal then the beliefs will be accurate. However if the subgoal is not achievable under all circumstances (possibly due to the lack of sensing actions) this does not mean that the plan generation fails. The generated plan might still allow the plan variable to be used even if the values may be invalid under certain circumstances.
leading to actions executed with unverified preconditions.

For example, an internal belief fluent corresponding to the fluent `freediskspace` would be an agent fluent called `i_freediskspace` and the planner is defined as *knowing* the value of `freediskspace` in a combined plan and world state if `i_freediskspace = freediskspace` in that world. Note when this equality holds in a given state it means that the believed value of `freediskspace` is correct in that state.

The condition `i_freediskspace = freediskspace` is effectively equivalent to the formula `freediskspace` existing in the $K_v$ knowledge base in the PKS planner (R. Petrick & Bacchus, 2002) which designates that the planner knows the value of a function.

For readability in the implemented planner, during plan construction, belief fluents are instantiated, they are named for the fluent or fluent proposition to which they refer — the name is prefixed with `i_` (indicating internal) and is followed by a string representation of the proposition or value it represents. The naming is just a convention and has no impact on how the fluent is used (just as in a standard programming language where variable name is inconsequential).

An internal fluent is distinct from an external fluent in the actions applicable to that fluent: internal fluents may be freely copied and assigned a value, and are not affected by any exogenous events. The state of internal fluents conform to *closed world assumption*, whereas the external world requires the open world assumption.

During the plan construction (or re-planning) phase, the planner can use different methods to populate these beliefs. There are 3 approaches that the agent can choose from in order to set the value of a planner belief variable.

- Population from a direct sensing action — such an action establishes a direct equality (which holds at the moment of observation) between the internal belief fluent and the external fluent. (The sensing plan example in the examples section shows direct sensing of the fluent `dbState` into the plan variable `i_dbState` and use of `i_dbState` as an action parameter.)

- Assignment of a constant value — in which case the belief will only be accurate in the contingency where the corresponding external world fluent has that value.
• Contingent assignment of a value depending on the outcome of one or more indirect sensing actions.

Depending on which population approach is used, the plan belief variable may or may not hold the same as the world fluent which the plan variable represents. If the plan variable is not populated accurately and the variable is used as an action parameter, an action may be executed with an invalid parameter. If the plan variable is used for action control the action may be executed under circumstances where its preconditions are not met. This is execution of actions with unverified preconditions (Morgenstern, 1987). If a fluent \( x \) which is not directly sensable under all contingencies the planner could still construct a plan to execute actions which have this precondition — and the action would at least succeed in those contingencies where the belief is accurate.

**Reasoning about this representation**

A goal to gain knowledge of the database internal state would be represented during plan construction (see subsequent chapter) as the subgoal \( i.dbState = dbstate \). The planner can achieve this knowledge goal by assigning \( i.dbState \) based on the output of appropriate sensing actions. Using this concrete concept of knowledge, knowledge goals may be formulated in terms of the state of the planner variables (and hence reasoned about using standard causal reasoning).

### 3.4.3 Plan with knowledge acquisition

In this example:

\[
(name: i.dbState, value: 0)
\]

\[
(name: job1, 
 command: "i.dbState = getDBerror"
 startConditions:)
\]

\[
(name: job2, 
 command: "repairDB i.dbState"
)
a database has an internal error condition represented by a numeric fluent \textit{dbState}. The goal is to repair the error (and bring the value of \textit{dbState} back to zero) using the \textit{repairDB} job. The repair job is only successful if its input parameter contains the error number of the database error. (If the parameter does not contain the error number the repair action has no effect). \textit{dbState} is not an automatic fluent so its value must be explicitly sensed. A direct sensing action \textit{getDBError} exists which is able to establish the error condition and populate it into any provided plan variable. The plan is able to achieve the goal by first populating the database error number into the \textit{i.dbState} plan variable using the \textit{getDBError} job. The sensed value \textit{i.dbState} is then used as a parameter to the \textit{repairDB} job which successfully repairs the specified error in the database, leaving the \textit{dbState} as zero. The procedural pseudo-code form for such a plan would be as follows:

\begin{verbatim}
begin
    i.dbState = exec “getDBError”
    exec “repairDB i.dbState”
end
\end{verbatim}

Figures 3.13 — 3.17 illustrate the evolution of the state of the plan and world during execution of the plan. In Figure 3.13 the status of both jobs is \textit{Initialised}. The plan variable \textit{i.dbState} is initially unset. In Figure 3.14 job1 execution is triggered (it has no start conditions) and the job state changes to \textit{Executing}. In Figure 3.15 job1 completes and the effects of the sensing action take effect (\textit{i.dbState} is populated with the value of \textit{dbState}). In Figure 3.16 all start conditions for job2 are true and execution of job2 is triggered using the \textit{i.dbState} parameter value of 3. In Figure 3.17 The \textit{repairDB} action completes. Since the parameter value used held the correct value of the error \textit{dbState}, the action is successful and the database is repaired resulting in \textit{dbState} = 0.

3.4.4 Plan using external medium to record sensing results — medical problem

In this benchmark example plan from D. S. Weld et al. (1998):
Figure 3.12: Bomb dunk — bomb defused.

Figure 3.13: Sensing and repair plan — initial state.
Figure 3.14: Sensing and repair plan — sensing action starts.

Figure 3.15: Sensing and repair plan — sensing action complete, plan variable populated with results of sensing action.
Figure 3.16: Sensing plan — repair action starts with the parameter value of 3.

Figure 3.17: Sensing and repair plan — repair action completes, db repaired and dbState = 0.
(name: i_todays.date, value: 0)

(name: job1,
    command: "stain"
    startConditions:)

(name: job2,
    command: "i_isBlue = inspect"
    startConditions: job1.status=Completed)

(name: job3
    command: "i_isInfected=isBlue"
    start condition: job2.status=Completed)

the goal is to determine whether a patient has an infection. A diagnostic stain action turns a culture blue if and only if the patient is infected. An inspect action determines whether the culture is blue or not. The goal is to determine whether the patient is infected (held in plan variable i_infected). The plan first executes the stain action which changes the culture (external medium) to blue if the patient is infected. The plan then executes the inspect action which senses whether the stain is blue or not and records the result in the plan variable i_isBlue. The plan then executes an assignment command to assign the value of the i_isBlue planner variable to the planner variable i_infected which represents whether or not the patient is infected. The procedural pseudo-code form for such a plan would be as follows:

begin
    exec "stain"
    i_isBlue = exec "inspect"
    i_isInfected = i_isBlue
end

Figures 3.18 — 3.21 illustrate the evolution of the state of the plan and world during execution of the plan for the case where the patient is infected. Figure 3.18 shows the initial state. In Figure 3.19 since the start conditions for job1 were true in the initial state, the
stain job has run and the culture colour has gone blue. In Figure 3.20 the start conditions for job2 have become true and the inspect sensing action has run. The action has sensed that the culture is blue and has populated the plan variable i.isBlue with the value true. In Figure 3.21 the start conditions for job3 have become true and the planner assignment command has assigned a value of true to i.infected

The execution for the case where the patient is not infected follows the same execution except the culture does not turn blue and i.isBlue and i.isInfected both end up populated with the value false.

![Plan](image)

Figure 3.18: Medical diagnosis of infected patient using external media — initial state.

### 3.4.5 Plan to determine combination of a safe by trying different combinations

In this plan for benchmark example from R. Petrick and Bacchus (2002):

(name:i_is1,value:null)
(name:i_is2,value:null)
(name:i_is3,value:null)

(name: job1,
Figure 3.19: Medical diagnosis of infected patient using external media — stain action executed

Figure 3.20: Medical diagnosis of infected patient using external media — inspect action executed
the goal is to determine the combination of a safe (and in this case have the value of the combination available in a plan variable (i\_combination) so that subsequently it could be communicated to another party). In this illustrative example the combination is 1, 2 or 3.

A parameterised test action checkCombo \(?n\) returns true if and only if the value passed in as the integer action parameter \(?n\) is the same as the value of the safe’s combination. The plan executes the test actions checkCombo 1, checkCombo 2, and checkCombo 3. If any of them returns true it sets the value of i\_combination accordingly. The procedural pseudo-code form for such a plan would be as follows:

begin
   exec “i_is1 = checkCombo 1”
   exec “i_is2 = checkCombo 2”
   exec “i_is3 = checkCombo 3”

begin
if ( i_is1 = true) then
    i_combination = 1
if ( i_is2 = true) then
    i_combination = 2
if ( i_is3 = true) then
    i_combination = 3
end

Figures 3.22 — 3.24 illustrate the evolution of the state of the plan and world during execution of the plan for the case where the safe combination (fluent \textit{comb}) is 2.

Figure 3.22 shows the initial state.

In Figure 3.23 since the start conditions for \textit{job1}, \textit{job2}, \textit{job3} were true in the initial state, the \textit{checkCombo 1}, \textit{checkCombo 2} and \textit{checkCombo 3} actions have run, \textit{i_is1} and \textit{i_is3} have been set to \textit{false} by the \textit{checkCombo 1} and \textit{checkCombo 3} actions and \textit{i_is2} has been set to \textit{true} by the \textit{checkCombo 2} action. In Figure 3.24 the start conditions for \textit{job5} have become true, the \textit{i_combination} = 2 has run and \textit{i_combination} holds the value 2 — the combination of the safe.

3.4.6 Plan with contingent knowledge acquisition using external media to record results and plan merge

In this example:

(name: \textit{i_isCorrupted}, value: \textit{null})
(name: \textit{i_corruptedId}, value: \textit{null})

(name: \textit{job1},
    command: "$\textit{i_isCorrupted} = \text{checkDBConsistency}$
    startConditions:)

(name: \textit{job2},
    command: "$\text{recordCorruptedToLog}$
    startConditions: \textit{job1.status=Completed, i_isCorrupted=True}$


Figure 3.21: Medical diagnosis of infected patient using external media — infected plan variable assigned

Figure 3.22: Safe combination plan — initial state.
Figure 3.23: Safe combination plan — checkCombo actions run

Figure 3.24: Safe combination plan — combination value plan variable set
knowledge acquisition is only performed under some contingencies. The plan addresses the goal of completing a database consistency check \textit{checkDBConsistency} and sending an alert to the support team indicating success or failure of the consistency check. On those contingencies where the database is corrupted the plan must also communicate the id of the corrupted record to the support team. The corrupted record in the database can be determined by performing the sensing action \textit{recordCorruptedToLog} which records the value of the corrupted record to a log file (fluent \texttt{log.contents}) and then reading the value of the corrupted record id from the log file using the action $i.corruptedId = \texttt{readLog}$. The representation allows information about the corrupted id to be transmitted to the plan via a recording of the value of the corrupted id in the external world. Plans which address different goals for different contingencies may need to be constructed if the job schedule forms part of an overall higher level plan (some of which consists of actions performed by the support team.) In this case the plan goals may achieve subgoals within this overall plan. The procedural pseudo-code form for such a plan would be as follows:

\begin{verbatim}
begin
  i.isCorrupted = null
  i.corruptedRecord = null
  i.isCorrupted = \texttt{exec "checkDBConsistency"}
  if ( i.isCorrupted = True ) then
    \texttt{exec "recordCorruptedToLog"}
    i.corruptedId = \texttt{exec "readLog"}
  endif
\end{verbatim}
exec alertSupport i_isCorrupted i_corruptedId
end

This contingent behaviour is achieved in this plan representation by stipulating a start condition of \( i_{\text{corrupted}} = True \) for the \( \text{recordCorruptedToLog} \) action and stipulating a start condition for \( \text{readLog} \) that the \( \text{recordCorruptedToLog} \) job is completed. Once the plan variable \( i_{\text{corrupted}} = False \) is set it is never subsequently changed — ensuring that \( \text{recordCorruptedToLog} \) and \( \text{readLog} \) are only executed on the needed contingencies.

Figures 3.25 — 3.27 illustrate the evolution of the state of the plan and world during execution of the plan for the case where the database is not in a corrupted state.

Figure 3.25 shows the initial state. In Figure 3.26 the \( \text{checkDBConsistency} \) job has run and \( i_{\text{isCorrupted}} \) is populated with the sensed value \( False \). In Figure 3.27 the start conditions \( \text{job1.status} = \text{Completed} \) and \( i_{\text{isCorrupted}} = False \) are met and the job4 \( \text{alertSupport} \) action has been triggered and executed with parameter values of \( i_{\text{isCorrupted}} = False \) and \( i_{\text{corruptedId}} = \text{null} \).

Figures 3.28 — 3.32 illustrate the evolution of the state of the plan and world during execution of the plan for the case where the database is in a corrupted state.

Figure 3.28 shows the initial state where the corrupted id is 8. In Figure 3.29 the \( \text{checkDBConsistency} \) job has run and \( i_{\text{isCorrupted}} \) is populated with the sensed value \( True \). In Figure 3.30 the start conditions \( \text{job1.status} = \text{Completed} \) and \( i_{\text{isCorrupted}} = True \) were met for job2 and the job2 \( \text{recordCorruptedToLog} \) action has executed and the value of the corruptedId has been logged into \( \text{log.contents} \). In Figure 3.31 the start condition \( \text{job2.status} = \text{Completed} \) is met for job3 and the job3 \( \text{readLog} \) has been executed and the value of \( \text{corruptedId} \) has been read from \( \text{log.contents} \) into \( i_{\text{corruptedId}} \). In Figure 3.32 the start condition \( \text{job3.status} = \text{Completed} \) for job4 has been met and job4 \( \text{alertSupport} \) has been triggered and executed with parameter values of \( i_{\text{isCorrupted}} = True \) and \( i_{\text{corruptedId}} = 8 \)
Figure 3.25: Contingent plan — db not corrupted, initial state.

Figure 3.26: Contingent plan — db not corrupted, i_isCorrupted assigned a value of False
Figure 3.27: Contingent plan — Support alerted that DB is not corrupted

Figure 3.28: Contingent plan — db corrupted, initial state.
3.5 Existing Time Representations and How the Plan Representation can Support Temporal Reasoning

The handling of time for planning and scheduling is a significant field within its own right and there are many specialised and highly efficient representations and algorithms for reasoning over both time point and time interval based representations. Temporal models exist to handle both conjunctions and disjunctions of the following temporal relations which may between two events A and B (Bellini, 2000; Allen, 1991):

- before: (event A end time is less than start time of event B)
- meets (event A end time = event B start time)
- overlaps (event A end time > event B start time and event A end time < event B end time)
- starts: (event A start time = event B start time)
- during: (event A start time > event B start time and event A end time < event B end time)
- finishes: (event A end time = event B end time)
- equals: (event A start time = event B start time and event A end time = event B end time)

However the domain requirements do not require reasoning about all of these relations and require only a limited subset which cover the ways in which time considerations arise in this domain:

- Initial state or goals involving time conditions, e.g. a goal state with the following conditions:
  
  \[ \text{jobAstate} = \text{Completed} \land \text{time} > 7pm \land \text{time} < 7.30pm \]

- Events which have time conditions involved in the trigger or effect conditions, e.g a job which has a start condition of \( \text{time} > 5pm \), or a job whose duration has time bounds
— e.g. completion time is > its start time + 1 hour (i.e. its minimum duration is 1 hour).

- Causal relations between events which imply temporal relations between events (job A starts after 8pm and job B doesn’t start until jobA completes so job B starts after 8pm)

- Threat protection via enforced temporal orderings between events.

The invented action logic (see chapter 4 for details) draws upon the dynamics of physical systems and defines all evolution of the system based completely from its current state. This means that trigger conditions need only specify conditions on a time point value, not an interval of time (since what happened in the past is not relevant), hence the representation uses a time point approach. Additionally, the agent is able to use an absolute dating system (such as that defined in Allen (1991) because modeled events occur on a computer where there is ubiquitous availability of the computer system clock time available for stamping.

In order to integrate time as naturally as possible into the partial order planning representation and inferencing, time conditions for goal state, initial state and event trigger and effect conditions are handled as far as possible as per any other fluent, however there is some special handling where the treatment of time needs to be distinguished

3.5.1 Differences between time and other fluents in terms of planning

- Unlike most numeric fluents which actions may increase or decrease, no actions exist which decrease time.

- Since time increases monotonically with temporal ordering, if there is a causal ordering (and hence temporal ordering) between events A and B then the end time for event A is lower than the start time for event B.

- Conversely, if event A has an effect condition time < 20 and action B has trigger condition time > 20, then B is ordered after A.

- Threats posed by the time effects of an event are automatically resolved due to the fact that the threatening event and the threatened event are by virtue of their time
conditions ordered with respect to each other (see the chapter 4 for details of the threat
detection performed by JobPlan). For example, if event $T$ has an effect condition that
time > 50 and action $B$ has a trigger condition that time < 20 which is supported by
event $A$ which has an effect that time < 10 then the threat posed by $T$ on the causal
support provided by $A$ is resolved since action $T$ is guaranteed to occur after event $B$.

- Action pre and postconditions for general fluents are assigned concrete values when
the postcondition is unified with a goal condition. However the bounds on the time
fluent for an action postcondition may be specified relatively with respect to the start
time of an event (where that start time may vary with each contingency).

3.5.2 Time conditions

Goal condition, initial state and event pre and post time conditions are specified in the same
way as per any other fluent. In the agent, the only explicitly imposed time relations required
take the form of absolute time constraints such as time > 6am, (relational conditions are
not allowed in defining pre and post conditions — e.g. time of event 1 < time of event 2).

In terms of power of the time representation for goals an expressive choice was made to
not support goals which involve a disjunction of temporal relations (such as that described
by Schwartz and Pollack (2004)). Scheduling requirements only require conjunctive time
based constraints such as time > value $\land$ time < value. Any non overlap requirements for
events are enforced via threat protection considerations.

There is a parameterised exogenous event timepasses which 'brings about' a higher time
value which has a general effect of the form:

\[
world.time = ?anytime \quad \text{where } anytime \text{ is a parameter which can take any value as needed.}
\]

3.5.3 Durative events

Durative events are represented by defining both trigger and effect time conditions which
relate the time difference between time value when the event is triggered and the time
value when the event effects take place. For example if an event which changes a batch
job action from status *Executing* to *Completed* takes exactly 60 minutes to take effect and the condition \( time = 5.10pm \) is proven to occur in the trigger state for that event then the condition \( time = 6.10pm \) is proven to occur in the effect state of that event.

Exact event durations are not generally known so, instead a bound on duration may be used. (Durations for events are available for most jobs from supplied definitions or based on historical data). If the duration of a job run is bounded for example by \( 10 < duration < 20 \), then if the following time:

\[
\text{time} > 4\text{pm}
\]

is shown to occur in the trigger state for a job execution event, then the following condition is proven to occur in the effect state:

\[
\text{time} > 4.10\text{pm}
\]

and if the following condition is shown to occur:

\[
\text{time} < 5\text{pm}
\]

then the following condition is proven to occur in the effect state:

\[
\text{time} < 5.20\text{pm}
\]

### 3.5.4 Causal orderings and temporal relations

Causal orderings between events (defined via the action job start conditions) implicitly define constraints between the start and end times of causally related events. Explicit orderings may also exist between events due to time conditions which hold in the trigger or effect state of those events. Any solution plan will require that on the trajectories of the initial plan and possible world state, orderings between events must be consistent with both the causal orderings and time condition based orderings. E.g. a report generation
event whose trigger state includes the time conditions $time < 5pm$ and $disk.space > 1MB$ and a disk clear event whose trigger conditions include the time condition $time > 7pm$ and whose effect is $disk.space > 1MB$. From the time conditions which hold in these events it follows that the report generation event occurs before the disk clear event. It is impossible to prove causal support from the disk clear event to the report generation event for the condition $disk.space > 1MB$ since this would require an ordering from the disk clear event to the report generation event — in direct conflict with the ordering entailed by the time conditions.
Figure 3.29: Contingent plan — db corrupted i_isCorrupted assigned a value of True
Figure 3.30: Contingent plan — job run to record corrupted id to log file

Figure 3.31: Contingent plan — job run to read corrupted id from log file
Figure 3.32: Contingent plan — Support alerted that DB is corrupted and provided the corrupted id
Chapter 4

PLANNING WITH THIS REPRESENTATION

This chapter describes how plans with this representation are synthesised using a new action logic.

4.1 Plan Generation from the Transition Function

In order to determine whether a given plan achieves a goal, an algorithm must answer the question — does a state belonging to the set of goal states \( G \) occur on all trajectories of all states in the set of possible initial states \( S_0 \) (where the initial state includes both the initial state of the plan and the state of the world). To answer this question, the dynamic evolution of each possible plan and world state is determined by applying the transition function.

For a dynamic system with Kripke structure \( K \), transition relation \( R(s, s') \) (where \( s' \) is an immediate successor state of \( s \)), a backward searching model based planner (Nau et al., 2004, App. C) algorithm such as that shown in Table 4.1 may be used by to check whether a goal state occurs on all trajectories of all states in \( S_0 \).

The algorithm starts with a current solution state set which consists of goal states. The algorithm then iteratively adds to this solution state set the set of all predecessor states \( s \) which are guaranteed to transition into one of the states \( s' \) in the current solution set. (Since model based planners use a state to action based plan representation, \( R(s, s') \iff s \rightarrow a \) is in the plan and where the transition relation \( R_a \) for \( a \) is the action which produces the transition \( R_a(s, s') \)). The algorithm iterates until this solution set contains all states in \( S_0 \) (in which case all states in \( S_0 \) are guaranteed to evolve into a state in \( G \)) or the solution set can no longer be expanded (in which case there is no plan which will evolve all states in \( S_0 \) into a state in \( G \)).
The algorithm is shown in Table 4.1

Table 4.1: Model Based Planner Search Algorithm

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCHECKAF(G, K)</td>
<td></td>
</tr>
<tr>
<td>currentSolutionSet = {}</td>
<td></td>
</tr>
<tr>
<td>nextSolutionSet = G</td>
<td></td>
</tr>
<tr>
<td>while nextSolutionSet ≠ currentSolutionSet</td>
<td></td>
</tr>
<tr>
<td>if (S0 ⊂ currentSolutionSet) then</td>
<td></td>
</tr>
<tr>
<td>textbf{return True}</td>
<td></td>
</tr>
<tr>
<td>currentSolutionSet = nextSolutionSet</td>
<td></td>
</tr>
<tr>
<td>nextSolutionSet = currentSolutionSet ∪ PRED(currentSolutionSet, K)</td>
<td></td>
</tr>
<tr>
<td>endwhile</td>
<td></td>
</tr>
<tr>
<td>return False</td>
<td></td>
</tr>
</tbody>
</table>

This predecessor state set of a current solution state set S is designated $PRED(S, K)$ and consists of the set of all predecessor states $s$ such that $\forall s' \cdot (R(s, s') \implies s' \in S)$. Model based checkers for such transition functions can be directly calculated using efficient representations of these sets of states using BDDs (Bertoli et al., 2001).

4.1.1 Models and transition function for reasoning with the JobPlan representation

Domain of discourse

The domain of discourse for planning includes states (which includes the plan state in addition to the state of the external world) and trajectories. The state is described in terms of the following restricted subset of first order logic:

- A functional fluent refers to an attribute of an object in the world or in the plan state where that attribute has a particular value — e.g. $db.state$, $i_safe.c$, $world.time$. The naming convention follows an object oriented approach for readability purposes but has no impact on the reasoning. For any given domain there is an enumerated set $F$ of fluents.

- Constant values such as 1, $Good$, $True$.

- The relations $<$, $>$, $!=$, $=$.
• An enumerated set of named events \( E \).

The fluent \textit{world.time} denotes the discrete time value held in a state. A state is modeled as a labeling function \( L(s, f) \) which maps every fluent \( f \) to a constant value \( v \). The function may be expressed as a set of ordered pairs \((f, v)\) where \( f \in F \) and \( v \) is a constant. So for example, the mapping \( L(s0) \) of a ground state \( s0 \) for a system which has two fluents \( x.v \) and \( y.v \) might be:

\[
L(s0) = \{(x.v, 1), (y.v, 2)\}
\]

with:

\[
L(s0, x.v) = 1 \quad \text{and} \quad L(s0, y.v) = 2.
\]

A ground trajectory is a sequence of ground states \( \{s0, s1, s2...\} \) where the \textit{world.time} value which holds in each ground state increases by 1 for each successor state.

\textbf{Transition function}

In this dynamics all events are triggered deterministically when certain state conditions (defined as the trigger conditions for that event) occur. These trigger conditions may be defined as a set of propositions \( CT(e) \) which may or may not hold for a given fluent mapping function \( L \). When an event \( e \) occurs its brings about the defined effects for this event (defined by a set of effect conditions \( CE \) where each \( ce \in CE \) provides a mapping for each fluent affected by the event).

An event is fully defined by defining the trigger conditions and the effect conditions. The frame assumption is used in all transitions — fluent mapped values are the same before and after any event transition — except for those fluent values which caused by the event effects or caused by the effects of some other event which has occurred.

For example for a ground state \( s \) with label \( L(s) = \{(x.v, 1), (y.v, 2)\} \) and an event \textit{assign3} whose effect conditions are to assign the value of 3 to \( y.v \), the resulting ground state \( s' \) from the transition due to occurrence of this event would have label function \( L(s') = \{(x.v, 1), (y.v, 3)\} \).
For all transitions the time value is incremented (discrete time model):

\[ L(s', world.time) = L(s, world.time + 1) \]

The overall effects of an event do not occur immediately in the successor state on the trajectory (as is the case for a Kripke structure transition relation). Events have a variable duration \( d \) between the time \( tt \) at which an event is triggered and the time \( tt + d \) at which the effects of the event take place. The duration of an event is not controllable by the planner but may be bounded for a given event. A deterministic Kripke transition relation \( R(s, s') \) where \( s' \) is an immediate successor state of \( s \) may be defined for an event \( e \) via the introduction of the following additional state fluents which control the duration between the event being triggered and its effects occurring:

- \( e.occurrenceNumber \) - which contains how many times the event \( e \) has occurred. This fluent is incremented when the event \( e \) occurs.

- A vector of duration fluents \( e.duration(1), e.duration(2), e.duration(3)\ldots e.duration(n) \) determines the duration of each occurrence of the event (a different duration element is required for each event occurrence since for exogenous events the same event may be triggered multiple times and may have a different duration each time it is triggered.)

- The fluent \( e.runtime \) describes how much time has passed since \( e \) was triggered. In the initial ground state \( e.runtime = -1 \) which indicates that the event is not currently occurring.

With the addition of these fluents for an enumerated set \( E \) of events the deterministic transition relation \( R(s, s') \) is defined as follows:

If \( L(s, e.runtime) = -1 \) and \( \forall ct \in CT(e) \cdot ct(L(s)) \) then:

\[ L(s', e.runtime) = 1 \land \]

\[ L(s', e.occurrenceNumber) = L(s, e.occurrenceNumber + 1) \]

where \( ct(L) \) is the proposition \( ct \) evaluated on the fluent values defined by \( L(s) \).

If \( L(s, e.runtime) < e.duration(e.occurrenceNumber) \land L(s', e.runtime) \neq -1 \) then:
\[ L(s', e.\text{runtime}) = L(s, e.\text{runtime} + 1) \]

If \( L(s, e.\text{runtime}) = e.\text{duration}(e.\text{occurrenceNumber}) \) then:
\[ L(s', e.\text{runtime}) = -1 \land \forall f \in \text{dom}(CE(e)) \cdot L(s', f) = CE(e, f) \]

Note the event effects must include conditions which negate one or more of the event trigger conditions so the event is not immediately re-triggered. Additionally at any given time only one event of the same type can occur. For all of these transitions no other changes due to \( e \) besides these occur (this does not preclude changes due to other events of different types - see below). Additionally for all events a restriction is made where there must be a fluent established in the event effects for which no other event establishes that fluent. This restriction is applied in order to ensure orderings between model states for its trigger and effects states (see the subsequent discussion of orderings).

For the previous example, the trigger condition for the \textit{assign3} is \( x.v = 1 \) and the conditions \textit{assign3.duration\_1} = 5 and \textit{assign3.runtime} = -1, \textit{world.time} = 0 hold in the initial ground state. If \( x.v = 1 \) occurs at time point 10 (by some other event), then the \textit{assign3} event is triggered at that time and at time point 11 \textit{assign3.runtime} = 1. At time point 12 \textit{assign3.runtime} = 2, at time point 13, \textit{assign.runtime} = 3. At time point 15 \textit{assign.runtime} = 5 which is equal to \textit{assign3.duration}(0) so \textit{assign3.runtime} changes to -1 and \( y.v \) changes value to 3.

Note during the period from when an event is triggered to when its effects take place, its it possible for state changes to arise from the effects of other events. An implicit domain constraint on the actual duration of each event occurrence ensures that events which have conflicting effects do not occur at the same time. For example if \textit{eventA} has effect \textit{file1.exists} = \textit{True} and \textit{eventB} has effect \textit{file1.exists} = \textit{False} then the event durations for each occurrence of \textit{eventA} and \textit{eventB} in any state will be such that these events can never complete at the exact same timepoint.

All state transitions (except for changes in time value) are a result of a triggered event \( e \).

With this definition of the ground state transition relation the dynamic model used
is deterministic. Given an initial world and plan state and where that state includes the
duration and runtime fluents for all events, a dynamic ground trajectory state sequence
consists of a single state for any given time value. Given an enumerated set $E$ of events, a
ground trajectory may therefore be considered as a function of the initial plan and world
ground state.

4.2 An Action Logic for this Transition Function

In order to determine if a given plan achieves a given goal, the action logic must be able to:

- Represent the initial (world+plan) state, goal state and intermediate states.

- Perform temporal projection using the described transition function to determine
whether any given state eventually occurs on the trajectory of the initial (world+plan)
state (in computational tree logic (Nau et al., 2004, Ch. 17) this would correspond to
the $AF$ formula).

- Reason using a succinct state representation which supports a condition based de-
scription allowing numeric fluents and equality and inequality operators. (Reasoning
about individual states with this transition function is not viable).

A number of action logics exist (see Mueller (2006) for a discussion of action languages
and logics). The action logic for JobPlan has some commonalities with existing logics such
as the flux action language (Thielscher, 1999) and the event calculus (Shanahan, 1997a)
but also some key differences created to handle some of the specific required features of the
domain and plan representation. The logic is described below in terms of the the syntax
models and logical inferences.

Since event durations are not under the direct or indirect control of the planner, even
if a planner triggers two events at the same time the planner cannot control the order in
which the effects of those two events occur. Explicitly representing the set of predecessor
ground states which are guaranteed to evolve into a given state would there require reasoning
about all possible durations of each event, each one of which would correspond to a ground
trajectory with a different number of interim ground states between the event start and end
states. In the case of unbounded durations the number of possible inter-leavings of events is exponential in the number of events.

The action logic therefore employs a partial order approach to abstract away from the variations in duration of events in order to determine \( PRED(\text{currentSolutionSet}, K) \). It follows the partial order inferencing used by classical partial order planners such as those described in D. Weld (1999) but with the ability to handle reasoning over different contingencies. Partial order based reasoning also has the advantage of increasing efficiency by abstracting away from irrelevant consideration of the different possible interleaving of events which do not interact in their effects.

At the action logic level if the fluent \( \text{world.time} \) is excluded from consideration the transition relation is deterministic and all uncertainty in the system is represented in the initial state. Each transition is described using an event definition which represents each transition symbolically by providing the symbolic state descriptor of all states which will trigger that event (trigger state) and the symbolic state descriptor of those aspects of the state which are changed by the event (the effect state description). In the resultant post transition state — all the conditions hold which held in the pre transition state — except those which are explicitly invalidated by the conditions which hold in the effect state. Additionally all of the effect conditions hold in the resultant state. Note this approach assumes that once the trigger state for an event occurs that nothing can prevent the effect state from occurring, even though the effect state occurs at a later time than the trigger state.

4.2.1 Syntax

The sentences of the action logic follow the grammar shown in Table 4.2.

4.2.2 Models

The models of this language are as follows:
Table 4.2: Action Logic Syntax

\[
\begin{align*}
\langle predicateSentence \rangle & := \langle predicate \rangle | \langle implication \rangle | \langle eventDefn \rangle \\
\langle predicate \rangle & := [-] \langle statePredicate \rangle | \langle trajectoryPredicate \rangle \\
\langle statePredicate \rangle & := \text{“Holds(“} \langle stateDesc \rangle \text{“, “, “condition“)}\text{“} \\
\langle eventDefn \rangle & := \text{“TriggeredEvent(“} \langle eventDesc \rangle, \langle stateDesc \rangle, \langle stateDesc \rangle \text{“)}\text{“} \\
\langle eventDefn \rangle & := \langle identifier \rangle \\
\langle trajectoryPredicate \rangle & := \text{“Occurs(“} \langle trajectoryDefn \rangle \text{“, “} \\
\langle trajectoryDefn \rangle & := \langle trajectory \rangle | \langle contingency \rangle \\
\langle trajectory \rangle & := \text{“Trajectory(“} \langle stateDesc \rangle \text{“)“} \\
\langle contingency \rangle & := \text{“Contingency(“} \langle stateDesc \rangle, \text{“)“} \\
\langle trajectoryPredicateDefinition \rangle & := \langle stateOn \rangle | \langle stateConditionOn \rangle | \langle protectionOn \rangle | \langle orderingOn \rangle | \langle threatOn \rangle \\
\langle stateOn \rangle & := \text{“StateOn(“} \langle stateDesc \rangle \text{“)“} \\
\langle stateConditionOn \rangle & := \text{“StateConditionOn(“} \langle stateDesc \rangle, \text{“} \\
\langle protectionOn \rangle & := \text{“ProtectionOn(“} \langle stateDesc \rangle, \text{“} \\
\langle orderingOn \rangle & := \text{“OrderingOn(“} \langle stateDesc \rangle, \text{“} \\
\langle threatOn \rangle & := \text{“ThreatOn(“} \langle stateDesc \rangle, \text{“} \\
\langle condition \rangle & := \text{“Condition(“} \langle valueTerm \rangle, \text{“} \\
\langle valueTerm \rangle & := \langle atomicTerm \rangle | \langle function \rangle \\
\langle atomicTerm \rangle & := \langle fluentName \rangle | \langle constant \rangle \\
\langle fluentName \rangle & := \langle identifier \rangle | \langle string \rangle \\
\langle function \rangle & := \langle fluentName \rangle (\langle + \rangle | \langle - \rangle) \langle constant \rangle \\
\langle constant \rangle & := \langle string \rangle | \langle number \rangle
\end{align*}
\]
Atomic terms

**constant** — A standard definition of constant which includes the real numbers, integers and string constants, e.g. 1, 1.3, *Completed*.

**fluentName** — the name of a fluent from the set $F$ of fluents.

**atomicTerm** — A term which is either a fluent, function or constant.

**valueTerm** — Consists of possibly a set of atomic terms, with each entry separated by ';'. This is used as a shorthand to denote a set of possible values for a given condition — e.g $1|2|4$ which indicates a value of either 1 or 2 or 4.

**stateDesc** — The name of a ground state. The state descriptor name is arbitrary and named for readability. Predefined names are reserved for a description of the current state `currentState` and goal state `goalState`.

Ground state predicates

**Condition(valueTerm, operator, valueTerm)** — Defines a predicate over ground states. The predicate is true for a ground state $s$ if the specified relation holds between the mapped value by $L(s)$ of the two specified value terms.

E.g. $\text{Condition}(x.value, <, 1)$ is true if the value mapped by $L(s)$ for $x.value$ is greater than 1.

$\text{Condition}(job1.status, ==, Completed)$ is true if the value mapped by $L(s)$ for $job1.status = \text{Completed}$.

**Holds(state, c)** — This predicate specifies a fluent condition $c$ which holds in all ground states which are a model of the state descriptor $state$.

E.g. the conditions which hold for the state descriptor `goalState` might be specified as:

$$\text{Holds}(\text{goalState}, \text{Condition}(\text{jobA.state} = \text{SUCCESS}))$$

$$\text{Holds}(\text{goalState}, \text{Condition}(\text{world.time} < 7\text{pm}))$$

$$\text{Holds}(\text{goalState}, \text{Condition}(\text{disk.freeSpace} \geq 5000))$$
A model ground state for goalState would be any state for which all of the these conditions hold, for example the ground state defined by fluent mappings:

\[
\text{jobA.state} \leftarrow \text{SUCCESS}, \text{world.time} \leftarrow 6.35\text{pm}, \text{disk.freeSpace} \leftarrow 7567
\]

The set of all such conditions which hold in a given named state stateDesc defines a function \(CS(stateDesc)\) which maps the state descriptor to the set of conditions c1,c2,c3.. which hold for that state descriptor.

**Zero place predicates**

**ConditionImplies(conditionA, conditionB)** states that for all ground states \(s\) for which \(conditionA\) holds then \(conditionB\) also holds in that ground state \(s\). E.g.:

\[
\text{ConditionImplies(Condition(x.v, >, 10), Condition(x.v, >, 5),)}
\]

I.e. \(\forall(\text{conditionB s.t. Holds(sb, conditionB)}).\text{Holds(sa, conditionB)}\)

since any ground state for which \(x.v > 10\) then \(x.v > 5\) is also true for that ground state.

**NegativeConditionImplies(conditionA, conditionB)** — has a similar definition to above except negated: for any ground state \(gs\) for which \(conditionA\) holds then \(conditionB\) does not hold in state \(gs\).

**StateImplies(sa, sb)** — states that for any ground state \(sm\) which is a model for state descriptor \(sa\) then \(sm\) is a model for state descriptor \(sb\).

**Trajectory predicates**

Note the parameter \(gt\) is dropped from all trajectory predicates since all reasoning performed in logic is performed over quantified sets of ground trajectories — see the subsequent discussion on the **Occurs** predicate.
**TriggeredEvent**($e$, $st$, $se$) — defines each event $e$ in $E$ the enumerated set of events by defining the set of trigger conditions $CT$ and effect conditions $CE$. $CT$ is the set of conditions defined for $st$ via the Holds predicate. $CE$ is the set of conditions defined for $se$ via the Holds predicate.

**Trajectory**($s$) - This predicate is used for reasoning about the (world + plan) evolution of different possible start states (note this is a different definition than the Trajectory predicate of the event calculus). Since the dynamics are deterministic (with time value and the event durations are encoded in the ground state description as described above), there is a single ground trajectory for a given initial ground state.

The definition ensures that the trajectory is consistent with the dynamics defined by $E$ that all changes are a result of the effects of a triggered event. Specifically $gt$ is a model for Trajectory($s$) if:

- The first ground state $gs0$ of $gt$ is a model for state descriptor $s$.
- $gt$ is a model for the dynamics defined by the set of events $E$.

**Contingency**($s$) is a predicate over ground trajectories. A ground trajectory is a model for Contingency($s$) if and only if it is a model for Trajectory($s$) and is also a model for Trajectory($currentState$). I.e. a model of a contingency is a ground trajectory of an initial ground state where additional conditions hold beyond those that are known to hold in $currentState$. Note that because the dynamics are deterministic contingency may be defined completely in terms of the initial state. From this point on contingency refers to a set of trajectories defined by the conditions which hold in the initial state.

**StateOn**($s$) is true for a ground trajectory $gt$ when a ground state which is a model for $s$ occurs on the trajectory.

**OrderingOn**($stateA$, $stateB$) is true for a ground trajectory when the first occurrence of any ground state which is a model for $stateA$ occurs before the first occurrence of any ground state which is a model for $stateB$.

**ProtectionOn**($stateA$, $stateB$, c) is true for a ground trajectory if:

- A ground state $gsa$ which is a model for $stateA$ occurs on the trajectory and is the
ground state which is the first occurrence of a model for stateA on the trajectory.

- For all ground states \( g_{ab} \) on the trajectory between \( g_{sa} \) and the first occurrence of any ground state which is a model for stateB the condition \( c \) is true for \( g_{ab} \), or if there is no occurrence of a model for stateB on the trajectory then \( c \) is true for every ground state \( g_{ab} \) on the trajectory after \( g_{sa} \).

\[ \text{ThreatOn}(st, \text{stateA}, \text{stateB}, c) \] — is true for a ground trajectory when:

- \( \text{Holds}(st, cn) \) where \( \text{NegativeConditionImplies}(cn, c) \)
- A ground state \( gst \) which is the first model for \( st \) occurs on the trajectory.
- A ground state \( ga \) which is the first model for stateA occurs on the trajectory.
- \( gst \) occurs after \( ga \) and before the first occurrence (if any) of a model for stateB.

\[ \text{StateConditionOn}(s, c) \] is true for a ground trajectory when there exists a ground state \( gsc \) on that trajectory where the condition \( c \) is true in \( gsc \) and where \( gsc \) is ordered on or before the first ground state \( gs \) model of \( s \) on the trajectory and for every ground state \( g_{ab} \) in the trajectory after \( gsc \) and before \( gs \), the condition \( c \) is true for \( g_{ab} \).

Note in terms of partial order planning (D. Weld, 1999), this corresponds to proving causal support for one of the individual preconditions \( c \) of an action node in the planning graph. Even though all the preconditions for the action may not have yet been proven, it is still necessary reason about the protections of \( c \) from the causal support for \( c \) to the node.

\[ \text{Occurs}(\text{Trajectory}(s), \langle \text{trajectoryPredicateDefinition} \rangle) \] is true when:

For every ground trajectory \( gt \) which is a model for \( \text{Trajectory}(s) \) then the specified \( \langle \text{trajectoryPredicateDefinition} \rangle \) is true also. (Note the quantified variable for \( gt \) is dropped for readability ).

The different forms of \( \text{trajectoryPredicateDefinition} \): StateOn, OrderingOn, ProtectionOn and ThreatOn are described above. For example \( \text{Occurs}(\text{Trajectory(currentState)}, \text{StateOn}(s)) \) states that a model ground state for state descriptor \( s \) occurs on all trajectories whose initial state is a model for the \( \text{currentState} \) state description. (Note this is equivalent to the CTL formula \( AFp \) where \( p \) is the propositional formula stating that each of the conditions which
All agent actions state transitions are characterised as using two or more TriggeredEvent predicates — one describing the start event of the agent action and one or more describing the action execution event. The start event axiom defines the event trigger state as a state which includes the condition that the action job for that action type is in *Initialised* state and the event effect state includes the condition that the action job for that action type has state = *Executing*. Other start conditions may be stipulated for an agent action start trigger state for control purpose. The action execution event(s) define the (possibly conditional) effects of the job. The trigger state definition for the nominal action execution event includes the condition that the job status for the action is in *Executing* state and other preconditions required for nominal execution. Non nominal execution events may also be defined where the preconditions for successful execution do not hold in the trigger state. The effect state definition of each execution event includes the condition that the job status for the action is *Completed*. The effect state also includes the external effects (nominal or nominal) of the action. Triggered event axioms and the state predicates for the trigger and effect state for agent actions are only asserted when an agent action is added into the plan. These axioms are asserted based on the definition of the action and the trigger and effect conditions instantiated specifically for that particular job (e.g. the job status trigger effect conditions) when it is added into the plan.

The state descriptor *st* which provides the description of the trigger state state is named

\[1\]

In the current implemented version of the plan dynamic model, since there is no event which resets the status of a job to *Initialised*, the start and complete events for the job only execute once. (If the plan required multiple executions of the same action, then there would need to be multiple jobs in the plan, each of which runs the same action). However for exogenous events it is possible for there to be multiple occurrences of the same event *e* if there is another event *r* which reinstates the trigger state of *e*. In order to handle this possibility and define a unique point in time for a given event and trajectory, the point in time of the event occurrence is defined as the first occurrence of a state which conforms to the event trigger state definition. In order to describe the trigger state of the second occurrence of a specific event *e* it is necessary to define a more specific event definition *e2* by specifying additional conditions which hold in the trigger state *e2* — such as time conditions or some fluent condition which indicates that the event *e* already occurred. Similarly, in order to define a specific point in time for the event effect on a given trajectory, the point in time of the event occurrence is defined as the first occurrence of a state which conforms to the event effect state definition. Since multiple events might have the same effect definition, in order to ensure that the first occurrence of an effect state *se* of a given triggered event *e* occurs after the first occurrence of any state corresponding to the trigger state *st* of that event, then it must be specified as part of the definition of *se* that the time value which holds in *se* is greater than the time value which holds in *st*. This time condition is not represented explicitly, it is encoded by ordering axioms between the trigger and effect states for each event.
(purely for readability purposes) with the event name + “.ts” and the state descriptor for the resulting state of the transition is named with the event name + “.es”.

For example a TriggeredEvent action execution axiom for a successful report batch job run event runJob_success triggered by action node 1 and which reads data from a message queue queue1 might consist of:

\[
\text{TriggeredEvent}(\text{runJob\_success}, \text{runJob\_success\_ts}, \text{runJob\_success\_es})
\]

and have the following state predicates defined for the nominal execution event trigger and effect states.

\[
\begin{align*}
\text{Holds}(\text{runJob\_success\_ts}, &\text{Condition}(\text{actionNode1.status} = \text{Executing})) \\
\text{Holds}(\text{runJob\_success\_ts}, &\text{Condition}(\text{queue1.state} = \text{GOOD})) \\
\text{Holds}(\text{runJob\_success\_es}, &\text{Condition}(\text{jobA.state} = \text{Completed})) \\
\text{Holds}(\text{runJob\_success\_es}, &\text{Condition}(\text{report.exists} = \text{True}))
\end{align*}
\]

the execution event definition for a failed batch job run might be:

\[
\begin{align*}
\text{TriggeredEvent}(\text{runJob\_failure}, &\text{runJob\_failure\_ts}, \text{runJobEffectState}) \\
\text{Holds}(\text{runJob\_failure\_ts}, &\text{Condition}(\text{actionNode1.status} = \text{Executing})) \\
\text{Holds}(\text{runJob\_failure\_ts}, &\text{Condition}(\text{queue1.state} = \text{BAD})) \\
\text{Holds}(\text{runJob\_failure\_es}, &\text{Condition}(\text{jobA.state} = \text{Completed})) \\
\text{Holds}(\text{runJob\_success\_es}, &\text{Condition}(\text{report.exists} = \text{False}))
\end{align*}
\]

Exogenous events are modeled as events triggered by conditions on fluents which are not internal agent fluents. For example the following event definition is for an exogenous event where an external process generates a file file1 at 10pm:

\[
\text{TriggeredEvent}(\text{fileGen}, \text{fileGen\_ts}, \text{fileGen\_es})
\]
Holds(fileGen_ts, Condition(world.time = 10pm))
Holds(fileGen_ts, Condition(file1.exists = False))
Holds(fileGen_es, Condition(file1.exists = True))
Holds(fileGen_es, Condition(world.time > 10pm))

Note, with this form of event definition the first occurrence of an event for a given
definition may happen at different times under different contingencies. This approach allows
the same event description occurrence to describe an outcome of different plan executions
under different contingencies. This allows the planner to reason about plans which have
remerging of different execution branches. For example, there may be two different initial
states one for which holds the condition Condition(reportFile.size < 100MB)) and another
initial state with Condition(reportFile.size >= 100MB). For the second situation the agent
may need to first run a split action on the file before ftping the files. In this case the same ftp
action job would be executed at different times for the two different contingencies, however
we would like to specify the ftp action event occurrence using a single job definition (to
avoid a combinatorial explosion of the number of event occurrences).

4.3 Axioms

This section presents the axioms for this action logic. Informal justification for the soundness
of these inferences follows from the interpretation, but formal proof of the validity of these
inferences is not presented here.

4.3.1 Causality and action effects

Proof of state occurrence on trajectory

A ground state which is a model for the state description of s occurs on all trajectories
whose start state is a model for s:

\[ \text{Occurs(StateOn(s), Trajectory(s))} \]

(AX1 (Start state of a trajectory occurs on that trajectory))
**State occurrence given condition occurrences**

A model for state $s$ occurs on a ground trajectory, iff a ground state $gs$ occurs in which is true every condition $c$ which holds in $s$ (defined by the function $CS(s)$ described previously):

\[
\forall c \in CS(s). StateConditionOn(c, s) \quad \iff \quad StateOn(s)
\]

(AX2 (State occurrence proven when all condition occurrences proven))

**State condition occurrence from causal support**

$StateConditionOn(c, s)$ occurs if there is a previous occurrence of a ground state $gsc$ in which $c$ is true and $c$ is protected from $gsc$ to $s$ From the definition of $StateConditionOn$ and the fact that any ground state for which $ca$ is true and where $ConditionImplies(ca, cb)$ then $cb$ is true in that ground state. In terms of partial order planner reasoning this inference is stating that in order to cause a condition in a given state — there must be a preceding model state for $sa$ which enables that condition and the condition $cb$ must not be disabled between $sa$ and occurrence of a model state for $sb$:

\[
(StateOn(sa) \land \exists ca \in CS(sa) \text{ s.t. } \exists ca \in CS(sa) \text{ s.t. } ConditionImplies(ca, cb) \land OrderingOn(sa, sb) \land ProtectionOn(sa, sb, cb) \iff StateConditionOn(cb, sb))
\]

(AX3 (Condition occurs if it holds in previous state and is protected))
State condition occurrence from occurrence of antecedent conditions

Simultaneous occurrences of conditions $ca$ and $cb$ in any ground state which is a model for $s$ where $ca$ and $cb$ imply $cc$ imply that $cc$ is true in that ground state:

$$StateConditionOn(ca, s))$$

$$StateConditionOn(cb, s))$$

$$∧(ca ∧ cb \implies cc)$$

$$\implies$$

$$StateConditionOn(cc, s))$$

(AX4 (Condition occurs if implying conditions occur))

(E.g. if the conditions $x = 1$ and $y = 1$ hold in a ground state, then the condition $x = y$ holds in that ground state.)

Event occurrence from occurrence of trigger state

The event occurrence inference states that the a ground state which is a model for the event’s effect state descriptor $se$ will occur on any ground trajectory where a model for the trigger state descriptor $st$ occurs.

$$TriggeredEvent(e, st, se)$$

$$StateOn(st)$$

$$\implies$$

$$StateOn(se))$$

(AX5 (Occurrence of event trigger state implies occurrence of event effect state))

4.3.2 Orderings between states

Orderings between model states of two state descriptors are proven using one of the following inferences:
Event effect state is ordered after its trigger state

For all events $e$, for any ground trajectory, the first occurrence of a ground state which is a model of the event effect state descriptor is ordered after the first occurrence of a ground state which is a model for the event trigger state. (Note this follows from the restriction that a triggered event effect must establish a fluent value for which no other event exists which establishes that fluent value).

$$TriggeredEvent(e, st, se) \implies OrderingOn(st, se)$$

(AX6 (Effect state of triggered event occurs after trigger state of event))

All states ordered after current state

During plan construction, the planner only reasons about future ground states, hence all states referred to in the logic are future states and as such any ground state is ordered on or any ground state which is a model of $currentState$:

$$\forall s : StateDescriptor
\quad OrderingOn(currentState, s) \quad \text{(AX7 (All states occur on or after current state))}$$

State occurs after all enabling events

State occurrence is ordered after the first cause of each of its underlying conditions since any predecessor state in which that condition does not hold.

- There is a state descriptor $sb$ in which condition $cb$ holds
- and all models for $sb$ are ordered all models for $sn$ and there is a condition $cn$ which holds in $sn$ and which implies $\neg cb$
- and there is an enabling event $fe$ which is ordered after $sn$ (i.e. all models for its trigger state are after all models of $sn$) and which enables condition $cb$
• and for all other events oe which enable cb either the event’s trigger state sto does not occur, or sto is ordered after the the effect state se of fe.

then sb is ordered after se:

\[
\text{OrderingOn}(sn, sb) \land cn \in CS(sn) \land
\]
\[
  cb \in CS(sb) \land
\]
\[
  \text{ConditionNegativelyImplies}(cn, cb) \land \text{TriggeredEvent}(fe, st, se) \land
\]
\[
  ce \in CS(se) \land
\]
\[
  \text{ConditionImplies}(ce, cb) \land
\]
\[
  \text{OrderingOn}(sn, st) \land
\]
\[
  \forall (oe \ s.t. \ (\text{TriggeredEvent}(oe, sto, seo) \land
\]
\[
  (ceo \in CS(seo) \land
\]
\[
  \text{ConditionImplies}(ceo, cb)))
\]
\[
.\neg \text{StateOn}(sto) \lor
\]
\[
\text{OrderingOn}(se, sto) \implies
\]
\[
\text{OrderingOn}(se, sb)
\]

(AX8 (First occurrence of a state is after every condition has been enabled))

**Transitive ordering**

If first occurrence of model ground state for sc occurs after first occurrence of model ground state for sb and first model for sb occurs after first model for sa, then first model for sc occurs after model for sa.
\[ \text{OrderingOn}(sa, sb) \land \text{OrderingOn}(sb, sc) \implies \text{OrderingOn}(sa, sc) \] (AX9 (Ordering transitivity))

**Time based ordering**

If state descriptor \( sb \) has a time condition \( cbt \) which involves a lower time bound whose value is greater than the upper time bound which holds in state \( sa \) then any model state for \( sb \) is ordered after model state \( sa \).

\[ cbt \in CS(sb) \land cbt = \text{Condition}(\text{world.time} > valueB) \land \]
\[ cat \in CS(sa) \land cat = \text{Condition}(\text{world.time} < valueA) \land \]
\[ valueA < valueB \implies \text{OrderingOn}(sa, sb) \] (AX10 (Ordering between states by time bounds))

### 4.3.3 Proving condition protections

**Protection if no threats**

A condition \( c \) is protected between the first occurrence of model for \( sa \) and the first occurrence of model for \( sb \) if and only if there are no ground states \( gts \) which occur between the \( sa \) occurrence and the \( sb \) occurrence which invalidate condition \( c \). This \( \text{ProtectionOn}(sa, sb, c) \) predicate is proven by identifying and disproving the occurrence of all such threat states.

The existence of a threat is represented by the predicate \( \text{ThreatOn}(st, sa, sb, c) \) which states that a ground state \( gst \) which is a model for \( st \) occurs between the first model state for \( sa \) and the first model state for \( sb \) and where condition \( c \) is false for \( gst \).
\( \text{ProtectionOn}(sa, sb, c) \)

\[ \Leftrightarrow \]

\[ \forall ts : \text{StateDescriptor} \land \neg \text{ThreatOn}(ts, sa, sb, c) \]

(AX11 (Protection if all threats disproven))

Disproof of occurrence of a threat state which is a model for \( ts \) follows from one of the following inferences:

**Threat resolved by demotion**

If first model ground state for \( ts \) occurs after first model ground state for \( sb \) then any threat posed by any ground state model for \( ts \) is disproved.

\[ \text{OrderingOn}(sb, ts) \]

\[ \Rightarrow \]

\[ \neg \text{ThreatOn}(ts, sa, sb, c) \]

(AX12 (Threat resolved by demotion))

**Threat resolved by promotion**

If \( ts \) occurs before \( sa \) on a trajectory then the potential threat posed by \( ts \) is disproved on that contingency.

\[ \text{OrderingOn}(ts, sa) \]

\[ \Rightarrow \]

\[ \neg \text{ThreatOn}(ts, sa, sb, c) \]

(AX13 (Threat resolved by promotion))
Threat resolved by contingency separation

If occurrence of any model ground state of threat state $ts$ is disproven on trajectory, then the threat is disproven on that trajectory

$$\neg StateOn(ts)$$

$$\implies$$

$$\neg ThreatOn(ts, sa, sb)$$

(AX14 (Threat resolved by separation))

4.3.4 Disproving occurrence of a state on a trajectory

Disprove state occurrence from disproof of state condition occurrence

In order to disprove the occurrence of a state $st$ on a trajectory, the simultaneous occurrence of all conditions which hold in that state must be disproven i.e. there must be at least one condition $uc$ which does not occur on the trajectory in the defined state.

$$(\exists uc \in CS(st)s.t.\neg StateConditionOn(uc, st))$$

$$\iff$$

$$\neg StateOn(st)$$

(AX15 (State does not occur if one of the conditions does not occur for that state))

Since occurrence of condition $ct$ in $st$ is proven from AX3 (Condition occurs if it holds in previous state and is protected), then state condition occurrence of $uc$ in in state $st$ is disproven by one of the following inferences:

Disprove state condition from disproof of supporting event

No supporting state for condition $uc$ — if $uc$ doesn’t hold in the initial state of trajectory and no events occur which cause $uc$. Since an event occurs on a trajectory if and only if the
event’s trigger state occurs on a trajectory, the event is disproven if and only if the event’s trigger state occurrence is disproven.

\[ Trajectory(si) \]
\[ cn \in CS(si) \land ConditionNegativelyImplies(cn, uc) \land \neg(\exists ee : State s.t. TriggeredEvent(e, et, ee) \land StateOn(et)) \land cc \in CS(ee) \land ConditionImplies(cc, uc) \land \implies \forall s : State \neg StateConditionOn(uc, s) \]

(AX17 (Condition occurrence disproven if no supporting events))

**Disprove state condition from ordering of the needed condition till after needed state**

Disproof of ordering between \( s \) and \( st \) for all states \( s \) which support \( uc \)

\[ \exists uc \in CS(st) s.t. \forall s : State s.t. StateOn(s) \land sc \in CS(s) \land ConditionImplies(sc, uc) \land OrderingOn(s, st) \implies \neg StateConditionOn(uc, st) \]

(AX18 (State disproven if one of the conditions support is always ordered after the state))

**Supporting events disabled**

Disproof of protection from all supporting states \( s \) to \( st \) of \( uc \)
\[\exists uc \in CS(st) s.t.
\forall s : State s.t. StateOn(s) \land sc \in CS(s) \land
ConditionImplies(sc, uc) \land
\neg ProtectionOn(s, sb, sc)
\implies
\neg StateConditionOn(uc, st))\]

(AX19 (Condition is disproven all causal supports are disabled by threat event))

4.3.5 Reasoning with sets of trajectories

To validate that a plan achieves a goal the occurrence of a state meeting the goal state definition must be proven on the trajectories of all states conforming to the current state definition. In order to reason efficiently about sets of trajectories in the implemented planner the inferences are defined using the \textit{Occurs} predicate which is a predicate over sets of trajectories (defined in terms of their initial state descriptions). The set of trajectories for which the consequent of any inference applies is determined by taking the intersection of the set of trajectories over which each of the antecedents is proven. For example, the implemented version of AX9 (Ordering transitivity) is:

\[\text{Occurs}(OrderingOn(sa, sb), Trajectory(sab)) \land \text{Occurs}(OrderingOn(sb, sc), Trajectory(sbc)) \implies \text{Occurs}(OrderingOn(sa, sc, Trajectory(sab \cap sbc))\]

(AX20 (Ordering is transitive))

Where \(sab \cap sbc\) is the state descriptor describing the set of states which conform to both descriptor \(sa\) and descriptor \(sb\).
The planner contains inferences regarding proven occurrences over different sets of trajectories:

**Occurrence proven for subset of trajectories**

\[ \text{Occurs}(\text{TrajectoryPredicate}, \text{Trajectory}(sa)) \land \text{StateImplies}(sb, sa) \implies \text{Occurs}(\text{TrajectoryPredicate}, \text{Trajectory}(sb)) \]

(AX21 (Occurrence proven for a state descriptor is proven for more specific state))

**Occurrence proven for union of trajectories**

\[ \text{Occurs}(\text{TrajectoryPredicate}, \text{Trajectory}(sa)) \land \text{Occurs}(\text{TrajectoryPredicate}, \text{Trajectory}(sb)) \land sc = sa \cup sb \implies \text{Occurs}(\text{TrajectoryPredicate}, \text{Trajectory}(sc)) \]

(AX22 (Occurrence proven for two state descriptors is proven for union))

Where \( sa \cup sb \) is the state descriptor describing the set of states which is the union of the states described by \( sa \) and \( sb \).

### 4.4 Plan Synthesis

The planner approach for reasoning with this action language is that advocated by Stone (1998); Shanahan (2000), of planning as an abductive inference process. The plan is synthesised using backwards inferences which make abductive choices about the contents of the plan. Once a choice about the plan contents has been made, the planner performs forwards inferences to determine whether the combined initial plan and world states eventually
evolves into the goal state.

The planner is implemented as a search over plan and state space with a solution found when the planner is able to prove that the goal state occurs on all possible trajectories of all possible initial (world + plan) states. The search follows a plan generation and verification approach to finding a solution. Candidate plans are synthesised and forwards inferences used to prove that the (world +plan) state evolves into a state which meets the goal conditions. i.e. proof of the occurrence of a goal state is proven from the ‘axioms’ of the initial plan and world state and the forwards inferences of the action logic. One approach would be to generate all possible candidate plans of a given size and then check each one. However due to the exponential number of plans with a given plan size, an exhaustive search is not a suitable approach to finding plans. The planner approach for reasoning with this action language is of planning as an abductive inference process. The planner performs a regression based search using abductive inferences to find plans and world states which will evolve into the goal state. It performs this search depth first, incrementally building up the set of initial (world+plan) states $PC$ on which the goal is proven. The top level algorithm is shown in Table 4.3, where the algorithm variables are defined as follows:

Table 4.3: Top level plan generation algorithm

<table>
<thead>
<tr>
<th>function generatePlan(goalState) returns Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Plan \leftarrow \text{Empty plan}$</td>
</tr>
<tr>
<td>$PC \leftarrow {}$</td>
</tr>
<tr>
<td>$UC \leftarrow CS$</td>
</tr>
<tr>
<td>while $UC \neq {}$</td>
</tr>
<tr>
<td>$PC, Plan = \text{proveGoalOnUC}(PC, UC, Plan, goalState)$</td>
</tr>
<tr>
<td>if goal proven on some contingencies $PC$ in $UC$ then</td>
</tr>
<tr>
<td>$UC \leftarrow (UC - PC)$</td>
</tr>
<tr>
<td>else if not possible to prove goal on any states in $UC$ then</td>
</tr>
<tr>
<td>break</td>
</tr>
<tr>
<td>endif</td>
</tr>
<tr>
<td>endwhile</td>
</tr>
<tr>
<td>if $(UC ={})$ then</td>
</tr>
<tr>
<td>return ${}$</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>Fail</td>
</tr>
<tr>
<td>endif</td>
</tr>
</tbody>
</table>
• $CS$ is the set of initial world states.

• $PC$ is the set of initial world states on which occurrence of the goal is proven (under the current plan).

• $UC$ is the set of initial world states on which occurrence of the goal is not yet proven (under the current plan).

• $proveGoalOnUC(\text{PC, UC, Plan, goal})$ is a function which adds new elements into the plan which will prove the goal occurrence on contingencies in $UC$ while not interfering with the existing plan's achievement of the goal on the contingencies $PC$. This requires that plan additions do not threaten existing causal supports provided by the current plan on the contingencies $PC$.

Table 4.4 shows the procedure $proveGoalOnUC$ where the algorithm variables are defined as follows:

Table 4.4: $proveGoalOnUC$ procedure

<table>
<thead>
<tr>
<th>function $proveGoalOnUC(\text{PC, UC, Plan, goalState})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SG \leftarrow \text{{goalState}}$</td>
</tr>
<tr>
<td>$SP \leftarrow \text{{}}$</td>
</tr>
<tr>
<td>while $\neg (SP \subset UC)$ and plan is usefully extendable</td>
</tr>
<tr>
<td>$SP = \text{determineCausalAntecedents(\text{PC, UC, Plan, SG})}$</td>
</tr>
<tr>
<td>if regression successful then</td>
</tr>
<tr>
<td>$SG \leftarrow SP$</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>backtrack last plan extension.</td>
</tr>
<tr>
<td>if no backtrack available then break</td>
</tr>
<tr>
<td>endif</td>
</tr>
<tr>
<td>endwhile</td>
</tr>
</tbody>
</table>

• $SG$ is the set of state descriptors whose occurrence needs to be proven on $UC$

• $SP$ is the predecessor set of state descriptors whose occurrence on a trajectory will with the current plan entail the occurrence of every state descriptor within $SG$ on that trajectory.

Table 4.5 shows the procedure $determineCausalAntecedents$
Table 4.5: determineCausalAntecedents procedure

```plaintext
function determineCausalAntecedents(PC, UC, Plan, SG)
    for each state sg in SG
        for each condition sc which holds in sg
            Choose causal support from:
            an existing state proven on a contingency compatible with UC,
            or an existing plan action only proven on a contingency which does
            not overlap UC
            or a new plan action,
            or an exogenous event,
            Ensure that the chosen plan modification does not threaten any existing
            needed occurrences, protections or orderings for previously proven
            contingencies PC
            Ensure protection and ordering from the chosen support state sp
            to sg on the needed contingencies UC.
            if able to prove protection and ordering from sp to sg without threatening
            proof on PC then
                SP ← sp ∪ SP
        endfor
    endfor
    return SP
```

4.4.1 Completeness

If there is a proof for the plan in the action logic the algorithm will find the plan. Completeness of the algorithm in this sense can be informally argued by employing similar arguments used for the model based planner MBP (Bertoli et al., 2001) and the partial order planner TWEAK (Chapman, 1985). In the MBP completeness proof, the planner performs a breadth-first regression based search. In the strong planning version of MBP all goal states are regressed back to exhaustively enumerate all possible predecessor states s which are guaranteed to evolve into any state s' in the goal state set SG if a suitable action a is applied to s. Since in MBP the plan consists of a state to action mapping, SA, the pair < s, a > can simply be added to SA to ensure this transition. This exhaustively enumerated set of state, action pairs which will transition into a state in SG is defined as the preimage(SG). In turn each state s present in the set of state, action pairs in preimage can then itself be regressed back to yield preimage(s) and those state,action pairs added to SA. During this regression process for any given state only one action is allowed to be
associated that state. Because the effect of actions is immediate and because for a given state only one action can be executed, the addition of new state, action mappings to the plan \( SA \) does not invalidate any transitions previously entailed by the plan. Therefore it is guaranteed that all states in \( SA \) are guaranteed to evolve into a goal state. Since the algorithm at each iteration identifies all possible immediate predecessors, if a solution plan exists, eventually the state-action set will include all possible initial states and the solution plan is provided by \( SA \).

In the case of the presented JobPlan planner, the algorithm is complicated by the fact that actions are not considered as having immediate effects. Therefore it is not possible to explicitly define an immediate predecessor state \( s \) which is guaranteed (under a suitable action) to evolve immediately into a state \( s' \) in the current set.

In order to employ such a completeness argument with the JobPlan, the concept of a predecessor state must be appropriately adapted. The predecessor of a set of states \( SG \) is defined in terms of a set of states \( SP \) which are causal antecedents to every state in \( SG \).

A causal antecedent of a set \( SG \) of states \( sg \) consists of a set \( SP \) of (world + plan) states \( sp \) such that:

\[
(\forall t: \text{Trajectory} (\forall (sp \text { in } SP) (\text{StateOn}(t, sp))) \implies \forall sg \in SG \text{ StateOn}(t, sg))
\]

I.e. if all the states \( sp \) in \( SP \) occur on a trajectory \( t \) then all states \( sg \) in \( SG \) also occur on \( t \).

The use of this definition of causal antecedents corresponds to a partial order planning approach. The set of states \( sp \) are the trigger states for a set of action nodes in the partial order plan which provide complete causal support for each action in the set of actions whose trigger states members of \( SG \).

If the planner search algorithm is able to determine all possible causal antecedents \( SP_1, SP_2..SP_N \) of \( SG \) then the algorithm can in turn iteratively enumerate all possible causal antecedents for each \( SP_1, SP_2..SP_N \). By continuing to iterate backwards, eventually if a solution plan exists, then one of these causal antecedents is the state set \( SI = si \) which contains a single initial state corresponding to the initial world state and the initial state of the solution plan.

To ensure completeness of this algorithm, the following conditions must be met:
• **determineCausalAntecedents**(PC, UC, Plan, SG) must identify all possible causal antecedents for a given state set SG. This can be argued using the “modal truth criterion approach” employed in the TWEAK completeness argument (Chapman, 1985). This criterion stipulates that a proposition p is necessarily true in a state s on a trajectory t iff two conditions hold: There is a state cs coincident to or ordered previous to s in which p holds and where for every event c whose effect state implies \( \neg p \ c \) is ordered before cs on the trajectory t or is ordered after s.

This criterion is enforced by AX3 (Condition occurs if it holds in previous state and is protected) and AX12 (Threat resolved by demotion), AX13 (Threat resolved by promotion), AX14 (Threat resolved by separation).

The search enumerates all possible forms of support state cs for a given proposition/condition. A subsequent section discusses this enumeration.

• New plan elements introduced in **determineCausalAntecedents** must not invalidate any previous causal supports provided by the existing plan elements. This can be ensured if either the threat posed by the introduced action is avoidable via promotion or demotion (as used in classical planning) or via context separation (contingent planning). The requisite context separation can always be achieved if sensing actions exist which delineate the contexts and where execution of those sensing actions does not interfere with the plan’s achievement of the goals.

### 4.4.2 Determining all causal antecedents

The causal antecedents for a state occurrence are identified using abductive inferences which for any given goal or subgoal g will try to prove g by generating new subgoals which if proven will prove g using one of the forwards inferences from the action logic. E.g. if there is an inference:

\[ P \land Q \implies R \]

(where P, Q, R are predicates) and there is a goal to prove R then subgoals would be created to prove P and Q. In JobPlan, subgoals are the expression of a predicate which needs to proved (or disproved). The ‘meta’ predicates NeedsProof\( (\text{predicate}) \) and
*NeedsDisproof*(predicate) are used to indicate that predicate needs to be proved or disproved.

Abductive inference rules match against these needed predicates to either generate further subgoals or to address the needed predicate directly via modifications to the plan. These modifications consist of the insertion of a new action job or modifying the start conditions of an existing job (which may include a condition on a new plan variable). Once a choice about the plan components has been made, the planner performs all applicable forward inferences to determine which state occurrences, orderings and protections occur on the trajectories of the different possible initial states.

To ensure the search is complete and any solution plan will be found, for each forwards inference, there need to be one or more corresponding abductive inferences of the form above which for a given $R$ are guaranteed to generate subgoals for any possible sets of antecedents $P, Q$ from which $R$ will follow. For most of the forwards inferences the planner has corresponding abductive inferences which guarantee to generate all viable antecedents, however for a small number of the forwards inferences there are some ‘obscure’ solutions which the implemented abductive inferences will not be able to generate — where this is the case this is noted in the descriptions below.

### 4.4.3 Proof procedure for determineCausalAntecedents

The backwards subgoaling inferences used to implement *determineCausalAntecedents* have been created to use the following proof procedure which addresses each of the action logic forwards inferences in order to be able to derive occurrence of all states in $SG$ from the proof of occurrence of all states in $SP$. In particular these inferences include an abductive inference AB16 (Use sensing action to set planner variable) which performs sensing in order to establish appropriate contingency control.

In the formula *NeedsProof* denotes the need to prove the truth of the specified formula.

**To prove a state occurrence on a trajectory**

To prove the antecedents the forwards inference AX2 (State occurrence proven when all condition occurrences proven) — attempt to prove coincident occurrence of all the conditions
which define that state:

\[
\text{NeedsProof}(\text{Occurs}(\text{StateOn}(s), \text{Trajectory}(s))) \iff \\
\forall c \in CS(s) \text{NeedsProof}(\text{Occurs}(\text{StateConditionOn}(s, c), \text{Trajectory}(s)))
\]

(AB1 (To prove state occurrence prove occurrence of all conditions))

**To prove the occurrence of a state condition**

To prove the antecedents of AX3 (Condition occurs if it holds in previous state and is protected) attempt to prove occurrence of a supporting state \(sa\) for the needed condition \(cb\) in state \(sb\) via one of the following abductive rules:

*Subgoal to prove occurrence of the trigger state for an agent action already present in the plan, where the effect state \(sa\) of that action directly or indirectly causes \(cb\). Subgoal to prove an ordering from \(sa\) to \(sb\) and protection of the supporting condition \(cb\) from \(sa\) to \(sb\).*

(AB2 (Condition support from existing action))

*Add a new action into the plan and subgoal to prove occurrence of the trigger state for that action where the effect state \(sa\) of that action directly or indirectly causes condition \(cb\). Add the action into the plan by adding the initialised job status to Initialised and defining the TriggeredEvent predicate for the new action. Subgoal to prove an ordering from \(sa\) to \(sb\) and protection of the supporting condition \(cb\) from \(sa\) to \(sb\).*

(AB3 (Condition support from new action))
Subgoal to prove occurrence of the trigger state for an exogenous event where that event effect state \( sa \) directly or indirectly causes the required condition \( cb \). Subgoal to prove an ordering from \( sa \) to \( sb \) and protection of the supporting condition \( cb \) from \( sa \) to \( sb \)

(AB4 (Condition support from new exogenous event))

Subgoal to prove an ordering from \( sa \) to \( sb \) and protection of the supporting condition \( cb \) from \( sa \) to \( sb \), where \( sa \) is a possible initial state which directly or indirectly causes the required condition \( cb \).

(AB5 (Condition support from initial state))

For whichever chosen supporting state \( sa \) subgoal to prove:

- on the needed trajectory.
- The on the needed trajectory.

To prove an ordering from one event to another

To address the antecedents for the forwards inference AX8 (First occurrence of a state is after every condition has been enabled) where a state condition is ordered after its first cause the planner uses one of the following abductive inferences. Note that since the planner does not have control of the dynamics of exogenous events, it is only able to control the dynamics of agent actions (by adding start conditions to that action):

Where an ordering is required between two agent actions \( A \) and \( B \) the planner can establish the ordering by adding the job completion status of job \( A \) (i.e. the condition \( \text{jobA.status} = \text{Complete} \) — which is an effect state of the \( A \) action) as a start condition for action \( B \). Since the initial state of all agent actions is Initialised this will establish the fact that this trigger condition for \( B \) does not hold in the initial state and since there are no other events other than the execution of action \( A \) which will set its status to Complete, this ensures that execution of \( A \) is the first cause of this condition thus ensuring that \( B \) occurs after \( A \).

(AB6 (Ordering by job dependencies))
If the planner is attempting to prove support from an exogenous event $e$ for a condition $ce$ which is a precondition for a successful execution of action $B$ and the fluent involved in $ce$ is automatically sensable the planner adds $ce$ to the start conditions for $B$.

(AB7 (Ordering from start conditions))

To address the antecedents of AX10 (Ordering between states by time bounds) which proves orderings between states from the time conditions which hold in those states:

If an ordering is required between a state $sa$ and state $sb$ where $sb$ is the trigger state of an agent action $B$, and $sa$ has a time condition $world.time \leq v$ then set a start condition of $world.time > v$ for action $B$.

(AB8 (Ordering from time start conditions))

To prove a protection from state $A$ to another state $B$

To address the antecedents of AX11 (Protection if all threats disproven) — subgoal to disprove each threat occurrence where a threat is an effect state which negates condition $c$.

\[
NeedsProof(ProtectionOn(sa, sb, c)) \implies \forall ts: StateDescriptor \text{ s.t. } TriggeredEvent(tt, ts).NeedsDisProof(ThreatOn(ts, sa, sb, c))
\]

(AB9 (To prove protection disprove all threats))

During plan construction there is an implicit assumption that for the actions not yet in the plan the corresponding action events do not exist and hence cannot be a threat. This assumption may be expressed as:

\[\neg \exists e\]

Where $e$ is the event for a action $a$ which is not in the plan.
If $a$ is added to the plan (and the corresponding action event $e$ instantiated) the previous fact that $\neg\exists e$ is no longer true. Retraction of this fact may cause retraction of facts proven from this. In particular, any new threats generated by $e$ might mean that previously proven state occurrences may no longer be proven.

**To disprove occurrence of a threat**

From AX12 (Threat resolved by demotion), AX13 (Threat resolved by promotion), AX14 (Threat resolved by separation) threats may be disproven via one of the following inferences:

- Prove the threat state $ts$ occurs after the supported state $sb$.

  (AB10 (Prove protection by threat demotion))

- Prove the threat state $ts$ occurs before the supporting state $sa$.

  (AB11 (Prove protection by threat promotion))

- Disprove occurrence of the threat state $ts$ on the trajectory where the protection is needed (context separation).

  (AB12 (Prove protection by threat separation))

**To disprove occurrence of a state**

To address disproof of threat state $ts$ on a trajectory from AX15 (State does not occur if one of the conditions does not occur for that state), a subgoal can be created to disprove occurrence of $StateConditionOn(ts, uc)$ where $uc$ is one of the conditions which holds in $ts$: 
\[ \text{NeedsDisProof}(\text{Occurs}(\text{StateOn}(ts), \text{Trajectory}(s))) \]
\[ \Rightarrow \]
\[ \exists c \in CS(s) \text{ s.t. NeedsDisProof}(\text{Occurs}(\text{StateConditionOn}(s, c), \text{Trajectory}(s))) \]

(AB13 (To disprove state occurrence disprove occurrence of one of its conditions))

**To disprove occurrence of a state condition**

To disprove \( \text{StateConditionOn}(ts, uc) \) for the antecedents of AX17 (Condition occurrence disproven if no supporting events) create subgoals to disprove that condition \( uc \) holds in the initial state and subgoals to disprove occurrence of all enabling events for \( uc \):

\[ \text{NeedsDisProof}(\text{Occurs}(\text{StateConditionOn}(ts, uc), \text{Trajectory}(s))) \]
\[ \Rightarrow \]
\[ \text{NeedsDisProof}(\text{Holds}(\text{currentState}, uc), \text{Trajectory}(s)) \land \]
\[ \forall (e \in Es.t.\text{TriggeredEvent}(e, et, ee) \land uc \in CE(e)) \]
\[ \text{NeedsDisProof}(\text{Occurs}(\text{StateOn}(et), \text{Trajectory}(s))) \]

(AB14 (To disprove condition occurrence disprove in initial state and all enabling events))

**To prove a state occurrence on some contingencies and disprove on others**

For a threat state \( st \) which is the effect of an agent action \( A \) and where the state needs to be proven on some contingencies \( C \) and disproven on other contingencies \( TC \):
Create a new propositional belief planner variable \( v \). Add a start condition to \( A \) that \( v = True \). \( v \) is specified as having the value False in the initial state. Subgoals are created to disprove occurrence of \( v = True \) on all contingencies \( TC \) where the threat needs to be disproved and to prove \( v = True \) occurs on the contingencies \( C \) where \( A \) is needed. If the planner is able to establish occurrence of the true belief fluent according to these subgoals then the threat state \( st \) will not occur on the protection contingencies \( TC \) and will occur on the non-protected contingencies \( C \).

(AB15 (Contingency control via planner variable))

Note that start conditions for a job may be introduced for two purposes — to control the contingencies in which that job is executed and or to control when the job is executed. These are orthogonal control requirements operating on different dimensions (either the time dimension or the initial state dimension(s)) of the dynamic system.

When specifying a start condition for contingency control purposes the planner always uses a planner variable for that contingency control, whereas for timing control the planner can choose to add a condition directly involving the fluent (assuming the fluent is automatically observable). Distinguishing the start conditions according to these different control requirements aids with plan readability. Since a planner variable can be assigned a value which is derived from any automatic or internal fluent value, for contingency control there is no loss of generality in this approach. Additionally contingency control is based off of the initial fluent values - whose values can be captured using a one-time assignment to belief fluents.

**To prove a true belief variable on some contingencies and false on others**

To prove that a true belief occurs on some contingencies \( C \) and disprove that a true belief occurs on other contingencies \( TC \):
Identify a sensing action which outputs true on contingencies in C and false on contingencies TC. Assign the belief fluent the results of the sensing action. Subgoal to prove protection of the sensed fluents from the initial state to the occurrence of the sensing action.

(AB16 (Use sensing action to set planner variable))

The current implementation limits this inference to the case where C and TC are non-overlapping and where both define convex regions in state space (i.e. can be described as a conjunction of fluent conditions) and where the regions defined by C and TC can be separated via the value boundary of a single fluent $c^2$

For a needed proof of a True planner variable $v$ occurrence on a contingency where condition $c$ is true and a needed disproof of $v = True$ on contingencies where $c$ is false and $c$ is automatically sensible, then the planner adds an action to the plan which assigns a value of True to the planner variable $v$ and adds a start condition of $c$ to the assignment action.

If $c$ is not automatically sensible and there is a propositional sensing action AS which only assigns an output value of True when condition $c$ is true then the planner will add that sensing action AS into the plan and populate $v$ from the output of the sensing action.

The completion of the sensing action is added to the start conditions for any agent action which reference the value of $v$. With the fact that the initially $v = False$, the fact that there are no other events which set $v = True$ and the fact that the AC will not return a true result on the TC, the trigger state of the threatening agent action $A$ on the protection contingency $TC$ will be disproven.

To disprove occurrence of an exogenous event

For disproof of an exogenous threat event:

\[^2\text{Note to generalise this inference to all possible TC and C would require: removing any overlap between TC and C by giving priority to whether A was added into the plan before or after the action which it is threatening (former actions in the plan take priority so that state occurrences already proven for a given plan are not retracted when adding new actions); splitting TC into multiple convex regions; creating a belief fluent $bp_i$ for each region $r_i$ in TC; determining one ore more sensing actions to separate each $r_i$ from C and then predicating action A upon any of these belief fluents being true.}\]
Choose a condition $ec$ which holds in the event trigger state and create subgoals to disprove $ec$ on the needed contingency.

(AB17 (To disprove exogenous event disprove one of the event trigger conditions))

Currently the implemented planner does not support any abductive techniques to prove the antecedent for AX18 (State disproven if one of the conditions support is always ordered after the state) and AX19 (Condition is disproven all causal supports are disabled by threat event). This means that the planner is not able to achieve protection from an event threat via a plan in which all trigger conditions for that event occur, but never simultaneously. To be able to generate such a plan the plan would need to be constructed in such a way that although conditions $ca$ and $cb$ may occur on the plan, they never occur simultaneously so any event triggered by $ca \land cb$ will never occur. This is essentially the problem of ensuring the safety condition $\neg ca \lor \neg cb$. Such constraints can be respected by ensuring that an event which enables $cb$ can only be triggered if $ca$ has previously been disabled and the condition $\neg ca$ is protected until the trigger state of any event which enables $cb$.

### 4.4.4 Handling contingencies in subgoals

Whenever the planner sets up a subgoal for any occurrence (be it state occurrence, state condition occurrence, protection occurrence etc) it must specify which contingencies it wants that occurrence to be proven (or disproven) on. (As mentioned previously the dynamic model is deterministic and contingency here is defined completely in terms of trajectories of initial state possible states). The planner makes no apriori assumptions about which contingencies may require different agent actions — so it initially sets up a top level goal to prove the goal state on all contingencies of the initial state:

\[ \text{NeedsProof}(\text{Occurs(StateOn}(\text{goalState}), \text{Contingency}(\text{currentState}))) \]

Whenever a subgoaling inference occurs to address a particular metapredicate NeedsProof or NeedsDisproof it always uses the needed contingency from the metapredicate when specifying the needed contingency for any subgoals created from that inference.

During \text{proveGoalOnUC}(PC, UC, Plan, goal) as the planner adds elements to the plan it may be able to prove goalState only on a particular set of contingencies which are a subset
of $UC$. Under such a situation the planner does not backtrack from this incomplete solution because modifications to a plan which proves the goal on some of the contingencies in $UC$ is still useful.

If $proveGoalOnUC(PC, UC, Plan, goal)$ is able to establish occurrence of $goalState$ on contingencies $UPC$ where $UPC \in UC$ then $proveGoalOnUC(PC, UC, Plan, goal)$ removes these proven contingencies $UPC$ from the unproven set: $UC \leftarrow (UC - UPC)$.

In the implemented planner this step is implemented using a control rule:

*When goal state is proven on some contingencies, determine the remaining unproven contingencies and subgoal to prove those.*

(CON1 (Determine the remaining unproven trajectories (if any) and try to prove those”))

In order to reason efficiently about these contingencies sets, the implemented planner has mechanisms for determining the set operations $\cap, \cup, -$ on sets of trajectories and creating descriptions of these sets using the $Trajectory$ predicate.
4.5 Evaluation of the planner implementation

The planner was evaluated on key illustrative examples from the domain, focusing on those scenarios which demonstrate particular abilities of the representation and action logic. The key criteria for evaluation was in demonstrating the expressiveness of the planner, timing data for measuring efficiency was not collected. For each example, a detailed description is given of the inferences and control logic the planner uses to solve each problem. Below is a description of each of these examples with a description of the features it demonstrates. In some cases the implementation was able to solve the problem and in some cases due to implementation limitations the planner was not able to generate the solution — in such cases the reasons for this are discussed.

- Handling exogenous events, durative action monitoring and triggered actions — A report script generates a report for a given date by processing an input file. The input file is only received after it has been generated by an exogenous external event. The report generation takes a variable amount of time and must be monitored for completion. Once the report is generated the report file is ftped to a remote server for use by an external job. This example demonstrates planning with exogenous events, triggered events and action execution and monitoring of durative actions. This example is fully implemented — the generated plan is included in the appendix.

- Planning with knowledge goals, knowledge use, merged plan branches — A database has an error state \texttt{dbState} which must be repaired, where the error value can be 1, 2 or 3. To check for internal database errors a test script \texttt{checkDb} can be run which takes as an argument the error condition \( e \) it is checking for and outputs \texttt{true} if the database has that error or \texttt{false} if it doesn’t. There is also a \texttt{repairDB} script which takes as an argument an error number \( e \) and repairs that error condition (or which has no effect if the database does not have that condition). Using these scripts, if a database has an error, the error condition may be determined using the \texttt{checkDb} script and once the error number is determined the \texttt{repairDB} script can be called with this error number to remove that error condition and return the database state to nominal. This example is fully implemented — the generated plan is included in
the appendix.

- Sensing using exogenous action to obtain knowledge and use of an external medium to record knowledge — a variant of the above example where a database has an internal error, but in this example the value of the database error is determined by an exogenous event and communicated to the planner via an external medium (email). This example is fully implemented — the generated plan is included in the appendix.

- Contingent planning with temporal goals. In this example the goal is to deliver a report by the agreed delivery time. Generation of the report is dependent on the presence of an input file which is created by an exogenous event. If the file is delivered late and the report results will not be ready by the agreed delivery time the plan executes an action to change the agreed delivery time so that the goal is still met. This example is not implemented, but an outline of inferences which could be used to generate a plan is described.

4.5.1 Handling exogenous events, action monitoring and triggered actions

In this example the goal is to produce a report file “remoteReport1220” on a remote server. A report generation batch job which takes a date parameter generates a report on the local server which has contents corresponding to the specified date. The process requires as input a file `inputFile` which is generated by an exogenous event. An ftp action exists which copies a specified file from the local server to the remote server under a new file name.

The definitions for these events are shown in Tables 4.6, 4.7 and 4.8. As discussed an object oriented naming convention is used to name fluents which correspond to attributes of an object (such as a file).

The first key inference is an abductive inference to provide support for the goal condition `remoteReport1220.exists = True` by adding into the plan a new action job to run the action `ftpToRemote` with the action parameter `?file = “Report1220”` and instantiating all its associated events (start event, success event, fail event). It then creates subgoals to prove that the ftp start event is triggered, that the local file `Report1220.exists = True` exists in the trigger state of `ftpToRemote`, that the condition `remoteReport1220.exists = True` is
Table 4.6: Start, Successful and Failure Event definitions for “genReport ?date”.

<table>
<thead>
<tr>
<th>Trigger conditions</th>
<th>Effect conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( status = \text{Initialised} )</td>
<td>( status = \text{Executing} )</td>
</tr>
<tr>
<td>( status = \text{Executing} \land ) ( \text{inputFile.exists} = \text{True} )</td>
<td>( status = \text{Completed} \land ) ( \text{report?date.exists} = \text{True} \land ) ( \text{report?date.contents} = \text{?date} \land ) ( \text{report?date.location} = \text{localServer} )</td>
</tr>
<tr>
<td>( status = \text{Executing} ) ( \land ) ( \text{inputFile.exists} = \text{False} )</td>
<td>( status = \text{Completed} )</td>
</tr>
</tbody>
</table>

Table 4.7: Start, Success, and Failure event definitions for ”ftpToRemote ?file”

<table>
<thead>
<tr>
<th>Trigger conditions</th>
<th>Effect conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( status = \text{Initialised} )</td>
<td>( status = \text{Executing} )</td>
</tr>
<tr>
<td>( status = \text{Executing} \land ) ( \text{?file.exists} = \text{True} \land ) ( \text{?file.location} = \text{localServer} \land ) ( \text{?file.contents} = \text{?contents} )</td>
<td>( status = \text{Completed} \land ) ( \text{remote?file.exists} = \text{True} \land ) ( \text{remote?file.location} = \text{remoteServer} \land ) ( \text{remote?file.contents} = \text{?contents} )</td>
</tr>
<tr>
<td>( status = \text{Executing} ) ( \land ) ( \text{?file.exists} = \text{False} )</td>
<td>( status = \text{Completed} )</td>
</tr>
</tbody>
</table>

protected from the successful ftp event effect state to the goalState and that the successful event effect is ordered before the goal state. Similar subgoals are created for the location conditions.

Because the existence of a file is considered as an automatically sensed fluent, the planner inserts a start condition of \( \text{Report1220.exists} = \text{True} \) for the ftpToRemote job (this becomes part of the trigger state definition for the ftp start event) — to ensure that the ftp command will not be executed until the condition \( \text{Report1220.exists} = \text{True} \) is true.

(Note the file contents is not an automatically sensed fluent so this cannot be inserted as a start condition for the action).

Using similar inferences the planner provides support for the condition \( \text{Report1220.exists} = \text{True} \) by inserting a new job and instantiating the start, success and failure events for "genReport ?date" with the parameter substitution \?date = ”1220”.

The planner achieves the required ordering between the report generation and ftp action by inserting an explicit planner ordering between the two agent actions by adding
Table 4.8: Event definition for exogenous event \textit{externalFileGen}

<table>
<thead>
<tr>
<th>Trigger conditions</th>
<th>Effect conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{inputFile.exists = False}</td>
<td>\textit{inputFile.exists = True}</td>
</tr>
</tbody>
</table>

\(\text{genReportJob.status} = \text{Complete}\) to the start conditions for \textit{ftpToRemote}.

Support for the \textit{inputFile.exists = True} trigger condition for \textit{genReportJob} is obtained from the exogenous event \textit{externalFileGen}.

Since the \textit{externalFileGen} trigger state has no conditions, using the inference AX2 (State occurrence proven when all condition occurrences proven) the planner is able to prove occurrence of the \textit{externalFileGen} trigger event on all trajectories.

From the occurrence of \textit{externalFileGen} trigger state, the occurrence of its effect state follows from AX5 (Occurrence of event trigger state implies occurrence of event effect state):

\[
\text{Proven(} \text{Occurs(} \text{StateOn(} \text{externalFileGen}\_es, \text{currentState}) \text{)} \text{)}
\]

From the fact that \textit{inputFile.exists = false} in the initial state and \textit{externalFileGen} is the only event which produces effect \textit{inputFile.exists = True} from AX8 (First occurrence of a state is after every condition has been enabled) it follows that:

\[
\text{Proven(} \text{Occurs(} \text{OrderingOn(} \text{externalFileGen}\_es)\text{,} \text{genReport1220}\_\text{start}\_\text{ts}) \text{)}
\]

And from the fact that there are no events which threaten the condition \textit{inputFile.exists = True} from AX11 (Protection if all threats disproven) the protection is proven from \textit{externalFileGen} to the trigger state for \textit{genReport}:

\[
\text{Proven(} \text{Occurs(} \text{ProtectionOn(} \text{externalFileGen}\_es)\text{,} \text{genReport1220}\_\text{start}\_\text{ts,} \text{inputFile.exists = True}) \text{)}
\]

From these and AX3 (Condition occurs if it holds in previous state and is protected)
follows:

Proven(Occurs(StateOn(genReport1220)_start_ts, currentState))

From the fact that genReport1220.state = Initialised in the initial state and genReport_start is the only event which produces effect genReport1220.state = Executing from AX8 (First occurrence of a state is after every condition has been enabled) it follows that:

Proven(Occurs(OrderingOn(genReport1220)_start_es),
 genReport1220)_success_ts)

And from the fact that there are no events which threaten the condition genReport1220.state = Executing from AX11 (Protection if all threats disproven) the protection is proven:

Proven(Occurs(ProtectionOn(genReport1220)_start_es),
 genReport1220)_success_ts, inputFile.exists = True)

From these and AX3 (Condition occurs if it holds in previous state and is protected) follows:

Proven(Occurs(StateOn(genReport1220)_success_ts, currentState))

From the fact that the job ftpToRemote has genReport1220.state = Completed as a start condition, this action is proven as occurring after genReport1220_success_es and occurrence of the trigger state for the successful ftp event is proven and from that occurrence of the goalState on all trajectories of currentState is proven.

The complete plan consists of the following:
The implemented planner was able to generate and validate this solution.

4.5.2 Planning with knowledge goals, use of the obtained knowledge and merged plan branches

This example is from the repair database error scenario previously described where the database error must be determined and the repair action called with the appropriate error. There is no action to directly determine the database internal error condition, instead the only available sensing command is “checkDB $i_{dbState}$” which checks whether the database has a particular error $i_{dbState}$ (a value of 1 or 2 or 3), where $i_{dbState}$ is a planner variable. The value of the error is then used in calling the repair action repairDB $i_{dbState}$. The knowledge acquisition part of this problem is analogous to the standard knowledge planning problem safe combination problem (R. Petrick & Bacchus, 2002). However this scenario also includes the use of the obtained information which the safe combination example does not involve.

The event schema definition for the internal plan variable assignment action assign, the checkDB command and repairDB action are shown in Tables 4.9, 4.10 and 4.11.

Table 4.9: Event definition for "assign $x$ $y$".

<table>
<thead>
<tr>
<th>Trigger conditions</th>
<th>Effect conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>status = Initialised</td>
<td>status = Completed ( \land (?x = ?y) )</td>
</tr>
</tbody>
</table>
Table 4.10: Event definition for \( \text{result} = \text{checkDB} \ e \).

<table>
<thead>
<tr>
<th>Trigger conditions</th>
<th>Effect conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{status} = \text{Initialised} )</td>
<td>( \text{status} = \text{Executing} )</td>
</tr>
<tr>
<td>( \text{status} = \text{Executing} )</td>
<td>( \text{status} = \text{Completed} )</td>
</tr>
<tr>
<td>( \land (\text{result} = \text{checkDB} \ e) )</td>
<td>( \text{result} = \text{dbState} = \text{error} )</td>
</tr>
</tbody>
</table>

Table 4.11: Start, Success and Failure Event definition for “\( \text{repairDB} \ i_{\text{dbState}} \)”.

<table>
<thead>
<tr>
<th>Trigger conditions</th>
<th>Effect conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{status} = \text{Initialised} )</td>
<td>( \text{status} = \text{Executing} )</td>
</tr>
<tr>
<td>( \text{status} = \text{Executing} )</td>
<td>( \text{status} = \text{Completed} )</td>
</tr>
<tr>
<td>( \text{dbState} = i_{\text{dbState}} )</td>
<td>( \land \text{dbState} = 0 )</td>
</tr>
<tr>
<td>( \text{status} = \text{Executing} )</td>
<td>( \text{status} = \text{Completed} )</td>
</tr>
<tr>
<td>( \text{dbState} \neq i_{\text{dbState}} )</td>
<td></td>
</tr>
</tbody>
</table>

In the initial state:

\[ \text{Holds} \left( \text{currentState}, \text{dbState} = 1 \mid 2 \mid 3 \right) \]

(where the "\mid" is a shorthand used by the planner to signify that the value held is one of the specified values)

The goal is to repair the database error:

\[ \text{Holds} \left( \text{goalState}, \text{dbState} = 0 \right) \]

\[ \text{NeedsProving} \left( \text{Occurs} \left( \text{StateOn} \left( \text{goalState}, \text{Contingency} \left( \text{currentState} \right) \right) \right) \right) \]

In order to provide support for the goal condition the planner adds a repair action into the plan. Since the successful repair action event includes the trigger condition \( i_{\text{dbState}} = \text{dbState} \), it creates a subgoal to prove that condition this occurs in the trigger state for the repair action. It specifies that this needs to be proven on all contingencies since the top level goal needs to be proven on all contingencies.\(^3\)

\(^3\)The sensing subgoal asserted in this plan is a contemporary sensing goal to ensure that \( i_{\text{dbState}} \) holds
The planner also stipulates that the $i_{dbState}$ initially has the value *null* (this initialisation is stipulated for all planner variables).

The planner uses an indirect support inference rule: AB3 (Condition support from new action) to address the subgoal. This rule provides causal support for condition $i_{dbState} = dbState$ by providing causal support for $i_{dbState} = 1$ from a new agent action `assign i_{dbState} 1` and applying this action if/whenever $dbState = 1$. (Simultaneous occurrence of the conditions $i_{dbState} = 1$ and $dbState = 1$ implies $i_{dbState} = dbState$).

The rule validates the viability of such an indirect approach by checking that the condition $dbState = 1$ is viable — there is either an contingency where $dbState = 1$ holds or an event which enables $dbState = 1$.

It asserts:

$$ProofInProgress(StateConditionOn(repairDB_success_ts, Condition(i_{dbState} = dbState)))$$

to indicate that a proof for this predicate is being addressed and creates the following subgoals:

$$NeedsProof(Occurs(StateOn(assign1_start_ts), Contingency(currentState)))$$

$$NeedsProof(ProtectionOn(assign1_es, repairDB_success_ts, Condition((i_{dbState} = 1), Contingency(currentState)))$$

$$NeedsProof(OrderingOn(assign1_es, repairDB_success_ts, (i_{dbState} = v)))$$

the value of $dbState$ at the time the repair job is run. This subgoal is distinct from sensing the initial value of the database error, since potentially $dbState$ could change between the initial state and the point at which the repair action is triggered. To specify sensing of the initial value of the database status value (such as that specified by the `handoff` sensing operator used by the puccini planner (Golden, 1997)) the subgoal would need to describe correspondence between the value of $i_{dbState}$ and the value which $dbState$ holds in the initial state. Such a correspondence requires use of a correspondence value $v$ which expresses the equivalence between fluent values at two different times: $NeedsProof(Occurs(StateConditionOn(repairDB_success_ts, Condition(i_{dbState} = dbState)), Contingency(currentState)))$

$$NeedsProof(Occurs(StateConditionOn(repairDB_success_ts, Condition(i_{dbState} = v)), Contingency(currentState + Condition(dbState = v))))$$
Contingency(currentState))

Note that the contingencies specified in the subgoals are taken from the subgoal which
this inference rule is addressing. Since the planner is attempting to determine the causal
antecedents for each subgoal, as far as possible it tries to only execute assign i_dbState 1
when/if dbState = 1. Since dbState = 1 is not an automatically sensable fluent, it cannot
simply add the condition dbState = 1 to the start conditions for assign i_dbState 1. It
instead creates a new control planner variable i_control_db.state_1 which represents the
planner variable which corresponds to the world condition dbState = 1. The truth of this
variable is added to the trigger condition for the assign 1 action.

Holds(assign1_start_ts, Condition(i_control_db.state_1 == true)

In order to achieve the correspondence between the control variable and the condition
dbState = 1, the rule sets up subgoals that this control planner variable is set to true
on contingencies where dbState = 1 and not set to true on contingencies where dbState = 1.\footnote{This approach makes the assumption that the initial value of the proposition dbState = 1 is being sensed and populated into i_control_db.state_1. A more general approach would be to create a contemporary sensing subgoal:

\textit{NeedsProof(Occurs(StateConditionOn(Condition(i_control_db.state_1, ==, true), assign1_start_ts), Contingency(currentState + Condition(db.state, ==, 1))))}

\textit{NeedsDisProof(Occurs(StateConditionOn(Condition((i_control_db.state_1, ==, true), assign1_start_ts), Contingency(currentState))})

However since the current planner does not support use of a proposition for the rhs operand of a condition, this form of subgoal could not be supported and hence the less general approach of subgoaling for different behaviours on different contingencies was used instead.}

In order to achieve the necessary ordering between the assign1 and repairDB action
the inference rule AB6 (Ordering by job dependencies) specifies a start condition for the
checkDB action that the assign1 action has completed.

NeedsProof(Occurs(StateConditionOn(Condition(i_control_db.state_1, ==, true), assign1_start_ts), Contingency(currentState + Condition(db.state, ==, 1))))

NeedsDisProof(Occurs(StateConditionOn(Condition((i_control_db.state_1, ==, true), assign1_start_ts), Contingency(currentState))})
Population of \textit{i\_control\_db\_state\_1} is achieved with the rule AB16 (Use sensing action to set planner variable). This rule is invoked when the value \textit{true} for planner variable needs to be proven on some contingencies and unproven on other contingencies. It adds an appropriate sensing action into the plan to populate the planner variable. The rule determines that the trigger state for the sensing action \textit{i\_control\_db\_state\_1 = checkDB 2} would match the contingency on which a true value of the variable is needed. Such a sensing action can therefore be used to populate \textit{i\_control\_db\_state\_1} which will set the value to \textit{true} on the needed contingencies and will not set it to true on the unneeded contingencies. The rule inserts completion of the sensing action as a start condition for the \textit{assign1} action to ensure that the sensing action is performed before \textit{i\_control\_db\_state\_1} is accessed. It then creates the subgoal to ensure the sensing action is performed at least on the contingency where a positive result would be needed.

\begin{verbatim}
NeedsProof(Occurs(StateOn(checkDB_1_start_ts),
            Contingency(currentState + Condition(dbState ==, 1))))
\end{verbatim}

The checkDB positive outcome event has a trigger condition of \textit{dbState = 1} so the planner sets up the subgoal to prove occurrence of this trigger condition. Using the rule AB5 (Condition support from initial state) the planner provides support for this condition from the contingency where \textit{dbState = 1}.

Since the \textit{i\_control\_db\_state\_1 = checkDB 1} action has no start conditions the inference AX2 (State occurrence proven when all condition occurrences proven) proves the occurrence of this action start on \textit{Contingency(currentState)}:

\begin{verbatim}
Proven(Occurs(StateOn(checkDB_1_start_ts), Contingency(currentState)))
\end{verbatim}

From the inference AX5 (Occurrence of event trigger state implies occurrence of event effect state)
Proven(Occurs(StateOn(checkDB_1_start_es), Contingency(currentState)))

Since the trigger state for checkDB 1 returning true includes the condition \(dbState == 1\) the positive outcome sensing action is proven on:

\[\text{Contingency}(currentState + \text{Condition}(dbState == 1))\]

and disproven on:

\[\text{Contingency}(currentState + \text{Condition}(dbState \neq 1))\]

From the fact that there are no actions in the plan which threaten \(i\_\text{control\_db\_state\_1} = \text{true}\) and assign1_start_ts is ordered after checkDB 1 action (due to the explicit start condition specified for assign 1). The occurrence of the assign 1 action is proven on the contingency where \(dbState == 1\).

From the fact that there are no actions in the plan which threaten the condition \(i\_\text{dbState} = 1\), the inference AX11 (Protection if all threats disproven) asserts:

\[\text{Proven}(	ext{ProtectionOn(assign1\_es, repairDB\_success\_ts, Condition((dbState = 1), Contingency(currentState))})\]

From the fact that the only event which causes \(i\_\text{dbState} = 1\) is the “assign1” action and this condition doesn’t hold in the initial state, the inference AX8 (First occurrence of a state is after every condition has been enabled) proves:

\[\text{Proven}(	ext{OrderingOn(assign1\_es, repairDB\_success\_ts, Contingency(currentState))})\]
From this and:

\[
\text{Proven} \left( \text{Occurs} \left( \text{StateOn} \left( \text{assign1} \_ \text{es} \right) \right), \right.
\]
\[
\text{Contingency} \left( \text{currentState} + \text{Condition} \left( \text{dbState} == 1 \right) \right) \]

the inference AX3 (Condition occurs if it holds in previous state and is protected) asserts:

\[
\text{Proven} \left( \text{StateConditionOn} \left( \text{repairDB} \_ \text{success} \_ \text{ts}, \text{Condition} \left( \text{i} \_ \text{dbState} == 1 \right) \right) \right)
\]

The subgoal:

\[
\text{NeedsProof} \left( \text{Occurs} \left( \text{StateConditionOn} \left( \text{repairDB} \_ \text{success} \_ \text{ts}, \right. \right.
\]
\[
\text{Condition} \left( \text{dbState} == 1 \right), \text{Contingency} \left( \text{currentState} \right) \right)
\]

is addressed by the rule AB5 (Condition support from initial state). This rule asserts:

\[
\text{Proven} \left( \text{Occurs} \left( \text{StateOn} \left( \text{currentState} + \text{Condition} \left( \text{dbState}, ==, 1 \right) \right), \right.
\]
\[
\text{Contingency} \left( \text{currentState} + \text{Condition} \left( \text{dbState}, ==, 1 \right) \right) \right)
\]

\[
\text{NeedsProof} \left( \text{Occurs} \left( \text{ProtectionOn} \left( \text{currentState} + \text{Condition} \left( \text{dbState}, ==, 1 \right), \right. \right.
\]
\[
\text{repairDB} \_ \text{success} \_ \text{ts}, \text{Condition} \left( \text{dbState}, ==, 1 \right), \text{Contingency} \left( \text{currentState} \right) \right)
\]

\[
\text{NeedsProof} \left( \text{Occurs} \left( \text{OrderingOn} \left( \text{currentState} + \text{Condition} \left( \text{dbState}, ==, 1 \right), \right. \right.
\]
\[
\text{repairDB} \_ \text{ts}, \text{Condition} \left( \text{dbState}, ==, 1 \right), \text{Contingency} \left( \text{currentState} \right) \right)
\]

The only threat to \( \text{dbState} == 1 \) is posed by the \text{repairDB} action, but that threat is already disproved (via threat demotion) since that action effect is ordered after \text{repairDB} \_ \text{success} \_ \text{ts}, so the inference AX11 (Protection if all threats disproven) is able to
assert:

\[
\text{Proven}(\text{Occurs}(\text{ProtectionOn}(\text{currentState} + \text{Condition}(\text{dbState}, ==, 1), \text{repairDB}\_\text{ts}, \text{Condition}(\text{dbState}, ==, 1), \text{Contingency}(\text{currentState})))}
\]

From AX7 (All states occur on or after current state) follows:

\[
\text{Proven}(\text{Occurs}(\text{OrderingOn}(\text{currentState} + \text{Condition}(\text{dbState}, ==, 1), \text{repairDB}\_\text{success}\_\text{ts}, \text{Condition}(\text{dbState}, ==, 1), \text{Contingency}(\text{currentState})))}
\]

and from:

\[
\text{Proven}(\text{Occurs}(\text{StateOn}(\text{currentState} + \text{Condition}(\text{dbState}, ==, 1)), \text{Contingency}(\text{currentState} + \text{Condition}(\text{dbState}, ==, 1))))
\]

\[
\text{Proven}(\text{Occurs}(\text{OrderingOn}(\text{currentState} + \text{Condition}(\text{dbState}, ==, 1), \text{repairDB}\_\text{success}\_\text{ts}, \text{Condition}(\text{dbState}, ==, 1), \text{Contingency}(\text{currentState})))
\]

\[
\text{Proven}(\text{Occurs}(\text{ProtectionOn}(\text{currentState} + \text{Condition}(\text{dbState}, ==, 1), \text{repairDB}\_\text{success}\_\text{ts}, \text{Condition}(\text{dbState}, ==, 1), \text{Contingency}(\text{currentState})))
\]

the inference rule AX3 (Condition occurs if it holds in previous state and is protected) asserts:

\[
\text{Proven}(\text{StateConditionOn}(\text{repairDB}\_\text{success}\_\text{ts}, \text{Condition}(\text{dbState} = 1), \text{Contingency}(\text{currentState} + \text{Condition}(\text{dbState}, ==, 1)))}
\]

Note the contingency of the asserted proof is the result of intersecting the contingencies for the supporting state, the protection and the ordering since the proven occurrence in the supported state depends on all of these.

From:
Proven(StateConditionOn(repairDB_success_ts, Condition(dbState = 1)),

Contingency(currentState + Condition(dbState, ==, 1)))

Proven(StateConditionOn(repairDB_success_ts, Condition(i_dbState = 1)))

Contingency(currentState + Condition(dbState, ==, 1))

the inference AX4 (Condition occurs if implying conditions occur) asserts:

Proven(StateConditionOn(repairDB_success_ts, Condition(i_dbState = dbState)),

Contingency(currentState + Condition(dbState, ==, 1)))

Again the proven contingency of the needed condition is the result of intersecting the proven contingencies for each indirect supporting condition.

From inference AX2 (State occurrence proven when all condition occurrences proven) the planner asserts:

Proven(StateOn(repairDB_success_ts,

Contingency(currentState + Condition(dbState, ==, 1)))

From AX5 (Occurrence of event trigger state implies occurrence of event effect state):

Proven(StateOn(repairDB_success_es,

Contingency(currentState + Condition(dbState, ==, 1)))

and from AX3 (Condition occurs if it holds in previous state and is protected) the following is proven:

Proven(StateOn(goalState,

Contingency(currentState + Condition(dbState, ==, 1)))
Proven(StateConditionOn(goalState, Condition(i.dbState = 1)),
Contingency(currentState + Condition(dbState, ==, 1)))

which from AX4 (Condition occurs if implying conditions occur) follows:

Proven(StateConditionOn(goalState, Condition(i.dbState = dbState)),
Contingency(currentState + Condition(dbState, ==, 1)))

and from AX2 (State occurrence proven when all condition occurrences proven) follows:

Proven(StateOn(goalState),
Contingency(currentState + Condition(dbState, ==, 1)))

Once the proof of the goal state has occurred the control rule CON1 (Determine the remaining unproven trajectories (if any) and try to prove those”) fires. In the body of this rule a calculation is performed to determine on which contingencies the goal has NOT been proven — it does this by negating the proven contingency:

Contingency(currentState + Condition(dbState, ==, 1))

and determines the remaining unproven contingencies as:

Contingency(currentState + Condition(dbState, !=, 1))

(As discussed in the the implementation section, the planner contains functions for operating upon contingencies such as negation, overlap, intersection, and consolidation).

All of the existing subgoals are de-scoped by limiting them to the proven contingency — so for example the subgoals:
\[ NeedsProof(\text{Occurs}(\text{StateOn}(assign1,s)), \]
\[ \text{Contingency}(\text{currentState})) \]
\[ NeedsProof(\text{ProtectionOn}(assign1_es, repair\text{DB}_success_ts, \]
\[ \text{Condition}((\text{dbState} = 1), \text{Contingency}(\text{currentState})) \]
\[ NeedsProof(\text{OrderingOn}(assign1_es, repair\text{DB}_success_ts, \]
\[ \text{Condition}((\text{dbState} = 1), \text{Contingency}(\text{currentState})) \]

are re-scoped to become:

\[ NeedsProof(\text{Occurs}(\text{StateOn}(assign1_ts)), \]
\[ \text{Contingency}(\text{currentState}) + \text{Condition}((\text{dbState} = 1)) \]
\[ NeedsProof(\text{ProtectionOn}(assign1_es, repair\text{DB}_success_ts, \]
\[ \text{Condition}((\text{dbState} = 1), \]
\[ \text{Contingency}(\text{currentState}) + \text{Condition}((\text{dbState} = 1)) \]
\[ NeedsProof(\text{OrderingOn}(assign1_es, repair\text{DB}_success_ts, \]
\[ \text{Condition}((\text{dbState} = 1), \]
\[ \text{Contingency}(\text{currentState}) + \text{Condition}((\text{dbState} = 1)) \]

For any subgoal which was marked as \textit{ProofInProgress} the new re-scoped subgoal is marked as \textit{ProofInProgress} also to indicate that it was already addressed. The descoping is an important step which must be performed in order to reflect the fact that it is only useful to prove \textit{assign1_es} on the contingency \textit{Condition}((\text{dbState} = 1)). During threat resolution it is critical that the planner knows on which contingencies an action is needed and on which contingencies it is not needed in order that it may successfully employ threat resolution via contingency separation. After re-scoping the existing subgoals the rule sets
up a new top level goal on the remaining unproven contingencies:

\[ NeedsProving(Occurs(StateOn(goalState, Contingency(currentState) Condition(dbState, ! =, 1))) \]

In order to address the need to prove the condition \( dbState = 0 \) in the goal state, the planner inference AB2 (Condition support from existing action) which seeks causal support from the \( repairDB \) action which is already in the plan.

It sets up a subgoal to prove occurrence of the \( repairDB \) success event on the contingency:

\[ Contingency(currentState) + Condition(dbState, ! =, 1) \]

In order to address the need to prove occurrence of \( i_dbState == dbState \) in the \( repairDB \) trigger state on \( Contingency(currentState) Condition(dbState, ! =, 1) \) the planner uses an indirect support inference AB3 (Condition support from new action) to address the subgoal. This rule provides causal support for condition \( i_dbState = dbState \) indirectly by providing causal support for \( i_dbState = 2 \) from a new agent action \( assign i_dbState 2 \) and applying this action if/whenever \( dbState = 2 \). As for the case for \( dbstate = 1 \) the planner creates a new control planner variable \( i_control_db.state_2 \) which represents the planner variable which corresponds to the world condition \( dbState = 2 \). The truth of this variable is added to the trigger condition for the \( assign 2 \) action.

In order to achieve the necessary ordering between the \( assign2 \) and \( repairDB \) action the rule AB6 (Ordering by job dependencies) specifies a start condition for the \( repairDB \) action that the \( assign 2 \) action has completed. This inference modifies the existing start condition of the \( repairDB \) action from:

\[ assign1.status = Completed \]

to:
assign1.status | assign2.status = Completed

The sensing action checkDB 2 is chosen to populate _i控制_db.state_2. Using similar forwards inferences as for the assign1 case, the planner proves that the assign2 action is proven to occur on the contingency:

\[\text{Contingency}((\text{currentState}) \text{Condition} (\text{dbState}, ==, 2)).\]

The ordering of the “repairDB _i_dbState_” command to after the assign2 action on Contingency((currentState) + Condition(dbState, ==, 2) is asserted by the inference AX8 (First occurrence of a state is after every condition has been enabled) on the basis that assign2.status = Initialised in the initial state and that the other start condition assign1.status = Completed does not occur on Contingency((currentState)Condition(dbState, ==, 2)).

From forwards inferences the occurrence of _i_dbState_ = dbState in the trigger state for “repairDB _i_dbState_” success event is proven on the contingency _dbState_ = 2

After the assign2 action is added into the plan the planner identifies the following possible threats:

\[\text{ThreatOn(assign2.success_es, assign1.success_es, repairDB.success_ts, Condition(i_dbState, ==, 1)}\]

\[\text{ThreatOn(assign1.success_es, assign2.success_es, repairDB.success_ts, Condition(i_dbState, ==, 2)\]}

The planner must resolve these threats, and does so via context separation:

---

5In order to build plans which involve branch merges, the planner is able to create these disjunctive start conditions for actions. The planner keeps a track of which start conditions it has introduced for each precondition of a successful job run and for which contingencies it has introduced the start conditions. In this example it records that the dependency on assign1.status was introduced to ensure an ordering from assign1 to repairDB on Contingency(currentState) + Condition(dbState, ==, 1) in order to support the condition _i_dbState_ = _dbState_. When introducing the dependency on assign2.status to ensure an ordering from assign2 to repairDB on Contingency(currentState) + Condition(dbState, ! =, 1) to address the same precondition _i_dbState_ = _dbState_ the planner determines that there is no overlap between these contingencies and it therefore takes the disjunction of these conditions for the start condition of the repairDB action which is associated with satisfying the precondition of _i_dbState_ = _dbState_. Because of the disjunction of these conditions, there are therefore two execution paths (assign1 executes or assign2 executes) which would trigger the execution of repairDB action.
From:

\[ NeedsProof(Occurs(ProtectionOn(assign1\_success\_es,\newline\quad repairDB\_success\_ts, Condition((InternalFluent)i\_safe\_c,==,1)),\newline\quad Contingency(currentState + Condition(i\_dbState,==,1))) \]

and:

\[ ThreatOn(assign2\_success\_es,\newline\quad assign1\_success\_es, repairDB\_success\_ts, Condition(i\_dbState,==,1)) \]

the planner uses the rule AB9 (To prove protection disprove all threats)

\[ NeedsDisProof(Occurs(ThreatOn(assign2\_success\_es,\newline\quad assign1\_success\_es, repairDB\_success\_ts,\newline\quad Condition(i\_dbState,==,1)),\newline\quad Contingency(currentState + Condition(i\_dbState,==,1))) \]

From AX17 (Condition occurrence disproven if no supporting events) the occurrence of \( dbState = 2 \) on the contingency:

\[ Contingency(currentState + Condition(db\_state,!=,2) \]

is disproven.

From this and AX15 (State does not occur if one of the conditions does not occur for that state) the occurrence of:

\[ StateConditionOn(Condition(db\_state,==,2), checkDB2\_correctValue\_ts) \]

is disproven on:
Contingency(currentState + Condition(db.state, ! =, 2))

From this and AX15 (State does not occur if one of the conditions does not occur for that state) the checkDB2_correctValue event is disproven on:

Contingency(currentState + Condition(db.state, ! =, 2)).

From this and AX17 (Condition occurrence disproven if no supporting events) and the fact that i_control_dbState_2 is initially null:

DisProven(Occurs(StateConditionOn(Condition(i_control_db.state_2, ==, True), assign2_start_ts), Contingency(currentState + Condition(db.state, ! =, 2))))

which by AX15 (State does not occur if one of the conditions does not occur for that state) leads to:

DisProven(Occurs(StateOn(assign2_start_es), Contingency(currentState + Condition(db.state, ! =, 2))))

which by AX14 (Threat resolved by separation) is proven:

Disproven(Occurs(StateOn(assign2_success_es), Contingency(currentState + Condition(i_dbState, ! =, 2)))

From the fact that assign 2 is disproven on the contingency:

Contingency(currentState + Condition(i_dbState, ! = 2))

the planner is able to disprove the threat posed by assign 2

DisProven(ThreatOn(assign2_success_es), assign1_success_es, repairDB_success_ts, Condition(i_dbState, ! =, 2))

This is proof by context separation (Pryor & Collins, 1996) where a threat is resolved by ensuring that the threatening action does not occur in the same contingency where the
threatened causal support is needed:

$$\text{ProtectionOn}(\text{assign1\_success\_es}, \text{repairDB\_success\_ts}, \text{Condition}(i\_dbState, ==, 1)),$$
$$\text{Contingency}(\text{currentState} + \text{Condition}(i\_dbState, ==, 1))$$

From this the proof of occurrence of the assign 1 event remains proven on:
$$\text{Contingency}(\text{currentState} + \text{Condition}(i\_dbState, ==, 1))$$

and hence:

$$\text{Proven}(\text{Occurs}(\text{StateOn}(\text{goalState},
\text{Contingency}(\text{currentState}) + \text{Condition}(dbState, ==, 1)))$$
remains proven. Similar inferences show that:

$$\text{ProtectionOn}(\text{assign2\_success\_es}, \text{repairDB\_success\_ts}, \text{Condition}(i\_dbState, ==, 2)),$$
$$\text{Contingency}(\text{currentState} + \text{Condition}(i\_dbState, ==, 2))$$

and

$$\text{Proven}(\text{Occurs}(\text{StateOn}(\text{goalState},
\text{Contingency}(\text{currentState})\text{Condition}(dbState, ==, 2)))$$

Which from contingency consolidation AX22 (Occurrence proven for two state descriptors is proven for union):

$$\text{Proven}(\text{Occurs}(\text{StateOn}(\text{goalState},
\text{Contingency}(\text{currentState})\text{Condition}(dbState, ==, 1|2)))$$

The rule CON1 (Determine the remaining unproven trajectories (if any) and try to prove those”) fires again and re-scopes the existing subgoals, including:
\[\text{NeedsProof} \left( \text{Occurs(} \text{StateOn}(\text{assign2_ts}) \text{)} \right),
\]
\[\text{Contingency}(\text{currentState}) + \text{Condition}((\text{dbState}! = 1))\]

which is re-scoped by intersecting it with the proven goal contingency:

\[\text{Contingency}(\text{currentState}) \text{Condition}(\text{dbState}, ==, 1|2))\]

to:

\[\text{NeedsProof} \left( \text{Occurs(} \text{StateOn}(\text{assign2_ts}) \text{)} \right),
\]
\[\text{Contingency}(\text{currentState}) + \text{Condition}((\text{dbState} == 2))\]

The remaining unproven goal is asserted:

\[\text{NeedsProving} \left( \text{Occurs(} \text{StateOn}(\text{goalState}) \text{)} \right),
\]
\[\text{Contingency}(\text{currentState}) \text{Condition}(\text{dbState}, !=, 1)
\]
\[+ \text{Condition}(\text{dbState}, !=, 2))\]

By adding the \textit{assign i_dbState 3} action into the plan this subgoal is addressed in a similar fashion as per the other contingencies and the goal state is proven on the contingency \textit{Contingency}(\text{currentState}) \text{Condition}(\text{dbState}, == 3)) and the mutual threats disproven via context separation as before.

By performing the AX22 (Occurrence proven for two state descriptors is proven for union) the planner asserts:

\[\text{Proven} \left( \text{Occurs(} \text{StateOn}(\text{goalState}) \text{)} \right),
\]
\[\text{Contingency}(\text{currentState}) + \text{Condition}(\text{dbState}, ==, 1|2|3))\]

and since \textit{dbState} = 1|2|3 holds in the current state:
The final plan is:

(name: i_dbState, value: null)
(name: i_control_db.state_is1, value: null)
(name: i_control_db.state_is2, value: null)
(name: i_control_db.state_is3, value: null)

(name: check1,
command: i_control_db.state_is1="checkDB 1",
status:Initialised, startConditions:)

(name: check2,
command: i_control_db.state_is2="checkDB 2",
status:Initialised, startConditions:)

(name: check3,
command: i_control_db.state_is3="checkDB 3",
status:Initialised, startConditions:)

(name: assign1, command: i_dbState=1,
command: i_dbState=1,
status:Initialised,
startConditions:i_dbState_is1=True)

(name: assign2, command: i_dbState=2,
command: i_dbState=2,
status:Initialised,
startConditions:i_dbState_is2=True)

(name: assign3 command: i_dbState=2,
command: i_dbState=3,
status:Initialised,
startConditions:i_dbState_is3=True)

(name:repairDB,
command: "repairDB i_dbState",
status:Initialised,
startConditions:assign1.status|assign2.status|assign31.status=Completed )

4.5.3 Sensing using exogenous action to obtain knowledge and use of an external medium to record knowledge

In this example the goal is the same — to repair the database error. However in this example the only sensing action for the database is diagnosis by the database team. The DBA team perform this diagnosis if their email inbox has the message “requestDiagnosis” (denoted in this example by the value of their inbox). The planner has an action which sends such a request message to the DBA team. The planner has an action to read its inbox and populate the read value into a planner variable.

The planner attains the goal by constructing a plan where the DBA diagnosis action is triggered by the planner by sending a diagnosis request. The results of the diagnosis are obtained by the planner by monitoring the email inbox and when it has a response from the DBA, reading the diagnosed value into the i_dbState planner variable. The event definitions for these are shown in Tables 4.12, 4.13, 4.14

This example demonstrates how sensing can be performed by an exogenous event which is triggered by the planner and how the planner brings about transmission of the exogenously obtained knowledge to the planner whereupon it may be used to perform an action (the database repair).

Table 4.12: Event definition for DBADiagnose

<table>
<thead>
<tr>
<th>Trigger conditions</th>
<th>Effect conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBAInbox.value = requestDiagnosis</td>
<td>PlannerInbox.value = dbState</td>
</tr>
<tr>
<td></td>
<td>PlannerInbox.hasMessage = True</td>
</tr>
</tbody>
</table>
Table 4.13: Event definition for *emailDBA*“requestDiagnosis”.

<table>
<thead>
<tr>
<th>Trigger conditions</th>
<th>Effect conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>status = Initialised</td>
<td>status = Executing</td>
</tr>
<tr>
<td>status = Executing</td>
<td>status = Completed ∧ <em>DBAInbox</em>.value = ‘requestDiagnosis’</td>
</tr>
</tbody>
</table>

Table 4.14: Event definitions for ?*readValue* = “*readInbox*”.

<table>
<thead>
<tr>
<th>Trigger conditions</th>
<th>Effect conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>status = Initialised</td>
<td>status = Executing</td>
</tr>
<tr>
<td>status = Executing PlannerInbox.hasMoreMessages = True</td>
<td>status = Completed ? <em>readValue</em> = PlannerInbox.value</td>
</tr>
<tr>
<td>status = Executing PlannerInbox.hasMoreMessages = False</td>
<td>status = Completed ? <em>readValue</em> = null</td>
</tr>
</tbody>
</table>

The goal is the same as the previous example:

\[\text{Holds}(\text{goalState}, \text{dbState} = 0)\]

\[\text{NeedsProving}(\text{Occurs}(\text{StateOn}(\text{goalState}, \text{Contingency}(\text{currentState})))\]

Initially the following conditions hold:

\[\text{Condition}(\text{PlannerInbox.hasMoreMessages}, ==, \text{true})\]

As in the previous example, the planner subgoals using inference to prove occurrence of the goal condition \(\text{dbStatus} = 0\) using the support event \(\text{repairDBsuccess}\) and sets up the subgoal:

\[\text{NeedsProof}(\text{Occurs}(\text{StateConditionOn}(\text{repairDB}_\text{success}_t) \text{Condition}(i\_\text{dbState} = \text{dbState}), \text{Contingency}(\text{currentState}))\]

This uses the inference AB3 (Condition support from new action) to seek support from
the event \textit{readInboxValue} success event effect \(i_{\text{dbStatus}} = \text{Planner.inbox}\) with the additional needed condition that \(\text{Planner.inbox} = \text{db.status}\) (which the inference recognises is achievable via the \textit{DBADiagnosis} task).

The planner places an agent action for \textit{readInboxValue} into the plan. To achieve the subgoal \(\text{Planner.inbox} = \text{db.status}\) inference AB3 (Condition support from new action) selects the \textit{DBADiagnose} success event and subgoals to prove occurrence of the trigger state for this successful diagnosis event. Since the trigger condition for the successful read action includes the condition \(\text{PlannerInbox.hasMessage} = \text{True}\) and the fluent \(\text{PlannerInbox.hasMessage}\) is an automatically observable fluent, when the planner adds the action condition \textit{readInboxValue} into the plan it includes \(\text{PlannerInbox.hasMessage} = \text{True}\) as a start condition for \textit{readInboxValue}.

When proving support from an agent action, the planner creates the standard subgoal that the supporting action is ordering before the supported action:

\[
\text{NeedsProof}(\text{Occurs(OrderingOn(}
\text{readInboxValuesuccess_{es}, repairDBsuccess_{ts}),}
\text{Contingency(currentState))))
\]

This is achieved via the inference AB6 (Ordering by job dependencies) which inserts the completion of the \textit{readInboxValue} action as a start condition to the \textit{repairDBsuccess} action.

\[
\text{NeedsProof}(\text{Occurs(StateConditionOn(}
\text{Condition(DBAInbox.value, ==, requestDiagnosis),}
\text{errorDiagnosedSuccess_{ts}), Contingency(currentState)})
\]

This subgoal is then addressed by another fire of AB3 (Condition support from new action) by matching the effects of the planner action \textit{sendRequestToDBA} to the needed condition \(\text{DBAInbox.value, ==, requestDiagnosis}\). The final plan is as follows:
(i_db.status:null)
(name: sendRequestToDBA
   command: "sendRequestToDBA"
   start condition
   status:Initialised
)

(name: readInboxValue
   status:Initialised
   command: "i_db.status = PlannerInbox.value"
   start condition: Condition(PlannerInbox.hasMessage,==,true)
)

(name: repairDB
   command: "repairDB i_dbState"
   status:Initialised
   start condition: readInboxValue.status,==,Completed)

4.5.4 Contingent planning with temporal goals

In this example the goal is to deliver a report by the time defined by $sla.deliveryTime$ whose initial value is 8pm. Generation of the end of day report requires dependency on the presence of the end of day marker file $eodMarker$ which is created by an exogenous event at the end of the business day (whose time value is $business.EODTime$). The latest possible end of day is 10pm. If the $eodMarker$ file is created late and the eod report results will not be ready by $sla.deliveryTime$ the plan executes the action $email support@downstream$ “new SLA is 11pm” which changes the value of $sla.deliveryTime$ to 11pm. The communication action only successfully changes $sla.deliveryTime$ if the communication is performed before the current value of the sla delivery time. The event definitions for these are shown in Tables 4.15, 4.16, 4.17

This example has not yet been implemented, this section provides an outline of the inferences the planner should use to generate a plan for this problem.
Table 4.15: Event definition for exogenous event setEODMarker

<table>
<thead>
<tr>
<th>Trigger conditions</th>
<th>Effect conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eodMarker.exists = False</code></td>
<td><code>eodMarker.exists = True</code></td>
</tr>
<tr>
<td><code>world.time = business.EODTime</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.16: Start, Successful and Failure Event definitions for “runEODReport”.

<table>
<thead>
<tr>
<th>Trigger conditions</th>
<th>Effect conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>status = Initialised</code></td>
<td><code>status = Executing</code></td>
</tr>
<tr>
<td><code>status = Executing ∧ world.time &lt;</code></td>
<td><code>status = Completed ∧</code></td>
</tr>
<tr>
<td><code>eodMarker.exists = True</code></td>
<td><code>eodReport.exists = True</code></td>
</tr>
<tr>
<td><code>world.time = runEod.startTime</code></td>
<td><code>world.time &lt; (runEod.startTime + 1)</code></td>
</tr>
<tr>
<td><code>status = Executing</code></td>
<td><code>status = Completed</code></td>
</tr>
<tr>
<td><code>eodMarker.exists = False</code></td>
<td></td>
</tr>
</tbody>
</table>

The goal and subgoal to prove goal occurrence is:

\[ \text{Holds}(goalState, eodReport.exists = True) \]

\[ \text{Holds}(goalState, world.time < \text{sla.deliveryTime}) \]

\[ \text{NeedsProving}(\text{Occurs}(\text{StateOn}(goalState, \text{Contingency}(currentState))) \]

Initially the following conditions hold:

\[ \text{Condition}(eodMarker.exists = false) \]

\[ \text{Condition}(eodReport.exists = false) \]

\[ \text{Condition}(\text{sla.deliveryTime} = 8\text{pm}) \]

\[ \text{Condition}(\text{world.time} = 6\text{pm}) \]
Table 4.17: Start, Successful and Failure Event definitions for "communicateLateSLA".

<table>
<thead>
<tr>
<th>Trigger conditions</th>
<th>Effect conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>status = Initialised</td>
<td>status = Executing</td>
</tr>
<tr>
<td>status = Executing ∧ world.time &lt; sla.deliveryTime</td>
<td>status = Completed ∧ sla.deliveryTime = 11</td>
</tr>
<tr>
<td>status = Executing world.time &gt;= 8pm</td>
<td>status = Completed</td>
</tr>
</tbody>
</table>

For causal support for the goal condition $\text{eodReport.exists} = True$ the planner inserts into the plan a $\text{runEODReport}$ action using the AB3 (Condition support from new action) inference.

For causal support for the goal condition $\text{world.time} < \text{sla.deliveryTime}$ the inference AB3 (Condition support from new action) decomposes the goal $\text{world.time} < \text{sla.deliveryTime}$ into the subgoals $\text{world.time} < 8$ and $\text{sla.deliveryTime} = 8$ ($\text{sla.deliveryTime} = 8$ is clearly a viable condition since it holds in the initial state).

The subgoal $\text{world.time} < 8$ can be addressed via two different forms of support — from the initial state or from the effect state of the $\text{runEODReport}$ event.

Causal support from the initial state cannot be used for causal support for $\text{world.time} < 8$ since the time effect of the $\text{runEODReport}$ threatens this causal support and the threat cannot be resolved by promotion or demotion since the goalState is ordered after the $\text{runEODReport}$ event.

The planner must therefore obtain causal support for $\text{world.time} < 8$ using indirect support from the $\text{runEODReport}$ with the additional requirement that:

$$(\text{runEod.startTime} + 1) < 8$$

to be proven to occur in the trigger state of $\text{runEODReport}$. (This condition may be simplified to the equivalent $\text{runEod.startTime} < 7$).

For the $\text{runEODReport}$ trigger condition $\text{eodMarker.exists} = True$ the planner seeks
support from the setEODMarker exogenous event. For the runEODReport trigger condition runEod.startTime < 7 the planner uses the indirect support inference to seek indirect support for this from the condition world.time = runEod.startTime which holds in the trigger state of runEodReport and by proving the condition world.time < 7 in the trigger state for runEodReport.

It is able to seek support for the condition world.time < 7 from either the initial state or the setEODMarker event. As before it is not able to use support from the initial state for this since the runEODReport is ordered after the setEODMarker event and to avoid threats from the setEODMarker event it must seek support from the setEODMarker event for the condition world.time < 7. In order to do this, it uses the indirect support inference to seek support for this condition from setEODMarker with the requirement to prove business.EODTime ≤ 7.

The planner seeks support for the condition business.EODTime ≤ 7, from the inference AB5 (Condition support from initial state) by assuming the contingency businessEODTime ≤ 7.

From this, the forwards inferences are able to prove the goal state on the contingency businessEODTime ≤ 7. The planner inference CON1 (Determine the remaining unproven trajectories (if any) and try to prove those") which asserts the goal:

\[
\text{NeedsProving} \left( \text{Occurs} \left( \text{StateOn} \left( \text{goalState}, \right. \right. \right. \right. \\
\left. \left. \left. \left. \text{Contingency} \left( \text{currentState} + \text{Condition} \left( \text{businessEODTime > 7} \right) \right) \right) \right) \right)
\]

For this contingency, the planner will not be able to achieve the goal using the previous support approach since it will not be able to establish the condition businessEODTime < 7 on this contingency. It instead uses the indirect support inference to address the goal condition world.time < sla.deliveryTime using indirect support from worldTime < 11 and sla.deliveryTime < 11. The sla.deliveryTime = 11 condition it establishes by adding to the plan the action communicateLateSLA. Since sla.deliveryTime is a condition which was used in the causal support on the contingency businessEODTime ≤ 7, there is an existing subgoal:
NeedsProof(ProtectionOn(currentState, goalState, sla.deliveryTime = 8,
Contingency(currentState + Condition(businessEODTime ≤ 7))).

The planner therefore seeks to disprove the threat of communicateLateSLA on the contingency businessEODTime ≤ 7. In order to do this it adds contingency control for the communicateLateSLA action with a control variable \( i \_\text{control} \_\text{is} \_\text{businessEODTime} \_\text{gt} \_\text{7} \).

It seeks to set \( i \_\text{control} \_\text{is} \_\text{businessEODTime} \_\text{le} \_\text{7} = \text{true} \) on the contingency businessEODTime ≤ 7, and false on all other contingencies.

The planner attempts to find a sensing action which distinguishes between these contingencies.

Since the exogenous event setEODMarker includes a trigger condition of worldTime > businessEODTime, it is effectively a sensing action for this condition. A positive outcome of the sensing action is indicated by the condition eodReport.exists = True.

If sensing of the condition worldTime > businessEODTime is coupled with the condition worldTime = 7, then sensing of the condition businessEODTime < 7 may be achieved. The planner inserts a sensing action which is triggered at 7pm and which returns false if eodReport.exists = True and true if the file does not exist. From indirect reasoning the planner determines that this action is able to sense whether businessEODTime > 7.

The population of \( i \_\text{control} \_\text{is} \_\text{businessEODTime} \_\text{gt} \_\text{7} \) is performed using this sensing action.

Forwards inferences show that \( i \_\text{control} \_\text{is} \_\text{businessEODTime} \_\text{gt} \_\text{7} \) is populated with true on contingencies where businessEODTime > 7 and is populated with false on contingencies where businessEODTime ≤ 7.

The communicateLateSLA will only be performed on those contingencies where businessEODTime > 7. Since businessEODTime < 10 is guaranteed and the duration of runEODReport is < 1 the report is guaranteed to be delivered by 11pm, so the goal will be met on this contingency. On contingencies where businessEODTime ≤ 7 the report file will be delivered by the initial sla.deliveryTime of 7pm and the goal is met.

The plan which the planner should generate is:
(i_control.is_businessEODTime_gt_7: null)

(name: checkLate
  status: Initialised
  command: "(i_control.is_businessEODTime_gt_7 = not (eodMarker.exists))"
  start condition: world.time > 7 )

(name: runEODReport
  command: "sendRequestToDBA"
  start condition: eodMarker.exists = true
  status: Initialised )

(name: communicateLate
  command: "communicateLateSLA"
  start condition: i_control.is_businessEODTime_gt_7 = true
  status: Initialised )
Chapter 5

PLANNER IMPLEMENTATION

This chapter describes the design, implementation and evaluation of the prototype JobPlan planner.

5.1 Planner Design

5.1.1 Choice of language

The general requirements for a choice of implementation language or tool were:

- Open source.

- Preferably Java based to allow integration of agent actions.

- Forwards (preferably backwards also) forms of inference.

- Ability to encode abductive inferences.

- Truth maintenance / logical retraction mechanism.

- Ability to specify search control.

The following languages/architectures were considered:

- CLIPS — interpreted production rule system. This system was rejected due to performance reasons which had been experienced in the prototype agent and the lack of an inbuilt backtrack mechanism, CLIPS is not Java based.

- MANDARAX — Backwards based reasoning based Java based rules system. Does not have forwards based rules
• JACK — BDI based integrated agent architecture — a high level Java agent architecture based on the belief desire intention agent model. The agent has established representations for agent beliefs and plans so using JACK to implement the proposed model would not be viable.

• JCHR — JCHR is an embedding of Constraint Handling Rules (CHR) in Java. Forwards based rules evaluation

• JTP — Java theorem prover — since specific abductive inference mechanisms were created for this representation, use of a theorem prover would be difficult.

• Jboss rules — Java based rules production system. A forwards based production rule system. Has a mechanism to support logical retraction.

Jboss rules (Proctor & Tirelli, 2007) was chosen as it supports all the requirements and offers the maximum flexibility. Drools rules is a rules engine which allows the definition of rules which fire when the rule antecedents match a set of objects which have previously been inserted into the working memory. The body of a rule performs assertions of other objects or call procedural code which inserts or retracts objects in the working memory, these insertions and retractions then in turn may cause activations or deactivation of other rules. Each inference in the action logic corresponding to a production rule.

In this application the objects asserted into the working memory are all of class ReasoningObject (please refer to the Backaus-Naur syntax previously described). Most of the rules in the implementation correspond to forwards and backwards inferences for the action logic. However there are also rules which implement the planner control mechanism such as detecting when to backtrack, determination of when the plan is complete etc.

Setting up a problem to be solved requires inserting into the working memory the TriggeredEventDefinitions for the domain (which requires defining the trigger and effect states for all agent and exogenous actions), the initial world state (using the Holds predicate), the goal state (using the Holds predicate for each goal condition) and the NeedsProof, Occurs, StateOn predicates for the occurrence of the goal state on all trajectories of the current state.
Drools contains an automated logical retraction facility which was used to perform truth maintenance for the forwards inferences. With this feature any consequences asserted in the rule body are automatically retracted when the supporting facts which matched the head of the original rule fire are retracted. Drools does not perform backwards reasoning so all abductive subgoaling is handled using hand-coded rules. Although requiring additional effort to encode these it does allow a high degree of control for controlling the ordering in which the abductive rules are called. Drools contains a salience control which allows the order in which rules fire to be specified using a numeric value.

5.1.2 Implementation of generatePlan(goal)

The initial goal which is asserted is

\[
\text{NeedsProof}(\text{Occurs(StateOn(goalState)}, \text{Contingency(currentState)}))
\]

The plan initially has no jobs and no planner variables. From this goal the planner proceeds to extend the plan according to the previously described abductive inferences.

A planner control rule detects whenever the goalState is proven to occur on some set of contingencies \(PC\). When it detects this new proof, it determines the set of remaining contingencies on which the goal is not yet proven and sets up a subgoal to prove the occurrence of the goal state on those remaining contingencies. (i.e. the \(UC \leftarrow (UC - PC)\) step in the algorithm).

Any existing subgoals already created and addressed by the planner which have contingencies which don’t overlap with the unproven contingencies \(UC\) are ‘de-scoped’ by reducing the needed contingency on those subgoals to those contingencies \(PC\) which are already proven — this ensures that the new subgoal created for the remaining unproven trajectories \(UC\) does not overlap in contingency with any of the existing subgoals.

For example if there is a top level goal:

\[
\text{NeedsProof}(\text{Occurs(StateOn(goalState)}, \text{Contingency(currentState)}))
\]

which via the abductive rules gives rise to subgoals:
\( NeedsProof(Occurs(StateOn(goalState), Contingency(currentSate))) \)

for which it generates subgoals:

\( NeedsProof(Occurs(StateConditionOn(goalState, ca), Contingency(currentSate))) \)

and

\( NeedsProof(Occurs(StateConditionOn(goalState, cb, Contingency(currentSate))) \)

If it is only able to prove:

\[
\begin{align*}
\text{Occurs}(\text{StateConditionOn}(\text{goalState}, \text{ca}, \text{Contingency}(\text{currentSate}+) \\
+ \text{Condition}(\text{worldTime} < 7\text{pm})) \land \\
\text{StateConditionOn}(\text{goalState}, \text{cb}, \text{Contingency}(\text{currentSate}))
\end{align*}
\]

from which follows:

\( \text{Occurs}(\text{StateOn}(\text{goalState}, \text{Contingency}(\text{currentSate} + \text{Condition}(\text{worldTime} < 7\text{pm})))) \)

then the goal state detection rule (de-scopes) the existing subgoals:

\[
\begin{align*}
\text{NeedsProof}(\text{Occurs}(\text{StateConditionOn}(\text{goalState}, \text{ca}, \text{Contingency}(\text{currentSate})))) \\
\text{NeedsProof}(\text{Occurs}(\text{StateConditionOn}(\text{goalState}, \text{cb}, \text{Contingency}(\text{currentSate}))))
\end{align*}
\]

to:

\[
\begin{align*}
\text{NeedsProof}(\text{Occurs}(\text{StateConditionOn}(\text{goalState}, \text{ca}, \\
\text{Contingency}(\text{currentSate}) + \text{Condition}(\text{worldTime} < 7\text{pm})))) \\
\text{NeedsProof}(\text{Occurs}(\text{StateConditionOn}(\text{goalState}, \text{cb}, \\
\text{Contingency}(\text{currentSate}) + \text{Condition}(\text{worldTime} < 7\text{pm}))))
\end{align*}
\]
Contingency(currentState) + Condition(worldTime < 7pm))

and adds the new top level goal to prove the goal on the remaining unproven contingencies:

NeedsProof(StateConditionOn(goalState, cb, Contingency(currentState + Condition(worldTime ≥ 7pm)))

The purpose of this descoping is to ensure that the subgoals for the contingencies already proven do not extend beyond the overall usefulness of that subgoal (which already has been shown to be limited to the contingencies contained within PC).

If this step isn't performed then when considering threat protection for any actions previously inserted into the plan for this subgoal, the planner will try to prove protection of those action executions on contingencies where it makes no contribution to the goal, hindering plan development for the remaining unproven contingencies.

5.1.3 Implementation of proveGoalOnUC(PC, UC, Plan, goal)

This planner function is implemented by stipulating the goal :

NeedsProof(Occurs(StateOn(goalState, Trajectory(suc)))

where suc is the state description for UC. The abductive rules to address this goal are fired and when the proof of goal occurrence is detected the function returns to the top level function generatePlan.

5.1.4 Implementation of determineCausalAntecedents(UC, Plan, SG)

This function is addressed by the abductive inferences which choose causal support for a given set of state occurrence subgoals on the specified contingencies. The rule for each subgoaling abductive inference takes the following general form:

when NeedsProving(R) ∧ ¬ProofInProgress(R) then
\textbf{assert} NeedsProving(P), NeedsProving(Q), ProofInProgress(R)

Where $P$ and $Q$ are the subgoals created to address $R$.

When the planner fires a rule which generates new subgoals in order to prove an existing subgoal, it marks the existing subgoal as in progress to prevent other subgoaling rules firing for the same subgoal. It does this by asserting a \textit{ProofInProgress} metapredicate for this subgoal.

The following “metaPredicates” are supported:

\texttt{NeedsProof, NeedsDisProof, ProofInProgress, ProofSuperseded, Proven, DisProven.}

Some of the abductive rules make choice about the construction of the plan:

- Adding in new action jobs (either external or internal commands)
- Adding a plan variable.
- Adding contingent or ordering control to existing planner actions.

After the plan is extended, the planner forward inferences to determine which states, protections, etc are proven and on which contingencies. Once the subgoals $P$ and $Q$ are proven the forwards inference is then able to establish $R$.

Ensuring that the chosen plan modification does not threaten any existing proven state occurrences is achieved since the (de-scoped) subgoals for these protections and orderings are still in force, so the necessary contingency control newly introduce actions is invoked. E.g. if there is a \textit{NeedsProof(ProtectionOn(sa, sb, PC)} for a trajectory $PC$ on which the goal was already proven and new action $A$ is added into the plan which threatens this protection, the abductive rule AB9 (To prove protection disprove all threats) which addresses all threats on a protection will create a subgoal to disprove the new threat on the contingency $PC$. A threat resolution rule will then be invoked. Once all the subgoals are proven (at least over some contingencies in $UC$) the function returns.

\textbf{5.1.5 Performing forwards inferences to show the plan achieves the goal}

The planner avoids irrelevant deductions by limiting most deductions to only occur if they address a particular subgoal. This is implemented by including the relevant metapredicates
*NeedsProof* and *NeedsDisProof* in the rule head pattern which describe the subgoals which the planner is attempting to prove.

\[
\text{when } \text{NeedsProving}(R), \text{Proven}(P), \text{Proven}(Q) \text{ then assert } \text{Proven}(R)\]

Note as additional plan elements are added previously proven state, protection, state condition occurrences may be temporarily unproven due to the following:

- Adding a new action into the plan may pose a threat to an existing action’s effects and may mean that previously proven state occurrences are no longer proven.
- Adding in a start condition for an action will mean the action does not occur unless the start conditions are true — hence previously proven state occurrences which were supported by this action may no longer be supported.

This requires that truth maintenance detect when previously proven facts about state, state condition, protection and ordering occurrences no longer hold. This achieved using the in-built drools logical retraction mechanism.

### 5.1.6 Logical retraction, inconsistency checking and backtracking

#### Logical retraction

In order to retract facts which are no longer true, the planner must support logical retraction of previously proven occurrences as the plan is extended. This functionality was implemented using the drools inbuilt logical retraction mechanism. This allows objects ‘logically’ inserted into the working memory by a rule to be automatically retracted when the antecedent conditions which activated and fired that rule are no longer true (and have been retracted). Hence the planner asserts all proven occurrences as a logical insertion — so they will be retracted if any of the supporting facts from which they were proven are no longer true.

Other sentences such as *NeedsProof*, *NeedsDisproof* statements about the initial plan state are not asserted logically, since we do not wish these to become retracted when the plan is modified — we keep those intact and commit to any pre-existing sub goals which
have given rise to the plan so far even if the facts which helped fire the subgoaling rule are no longer true.

Facts which need to be asserted using the Drools logical assertion (i.e. facts which could become unproven due to a plan extension)

- The proof or disproof of any occurrences (state, state-condition, protection, ordering)

Facts which should not be asserted using the Drools ‘logical’ assertion (i.e. which should never be retracted due to a plan extension)


- Statements about the components in the plan — e.g. holds in the initial state that plan job1 command is ”assign 1 x”.

- Statements about the dynamics of the plan — e.g. conditions which hold in a particular event trigger state.

**Inconsistency detection**

Deduction rules which identify inconsistencies are marked with a high salience value (see Proctor and Tirelli (2007)). This means that deduction rules whose antecedents are true will fire before other rules who antecedents may also be true but whose salience value is lower. This allows inconsistencies to be identified immediately — before any further subgoaling takes place.

The inconsistency rules detect the following situations.

- Two subgoals which advocate that conditions hold in the same state and where those conditions are incompatible with each other.

- A subgoal to prove a temporal ordering from A to B and to prove an ordering from B to A.

- A subgoal to prove that an event occurs before the initial state.

- A subgoal to prove a predicate and a subgoal to disprove the same predicate.
Backtracking

Whenever a search dead end is reached (no more forwards or abductive inferences can be made), or an inconsistency is detected, the planner backtracks. Since all deductions (including the inconsistency deductions) have higher salience than the subgoaling rules, any bad subgoaling choices are immediately detected. This means that it is the last abductive choice which caused the inconsistency and hence backtracking to just before that subgoal decision will remove the inconsistency. The planner retracts the last abductive inference it made (addressing subgoal $sg$) and retracts all inferences which were made chronologically after that choice. It then chooses another abductive choice to address $sg$.

When any metapredicate is asserted, the time-stamp of its assertion is recorded. This makes it trivial to retract any facts which were asserted subsequently to any given subgoal. The planner keeps track of abductive rules which it has already made for a given subgoal to prevent the same abductive rule being re-fired for the same subgoal after backtracking has occurred.

5.1.7 Inference control

Prevention of circular contingency equivalence inferences

There are cases where circular inferences may occur — for example if:

\[
\text{Proven}(\text{Occurs}(\text{StateOn}(stateA), \text{Contingency}(\text{currentState})))
\]

then from inference AX21 (Occurrence proven for a state descriptor is proven for more specific state) the following are proven:

\[
\begin{align*}
\text{Proven}(\text{Occurs}(\text{StateOn}(stateA), \\
\text{Contingency}(\text{currentState} + \text{Condition}(x > 10)))) \\
\text{Proven}(\text{Occurs}(\text{StateOn}(stateA), \\
\text{Contingency}(\text{currentState} + \text{Condition}(x <= 10))))
\end{align*}
\]
However, likewise if:

\[
\text{Proven}(\text{Occurs}(\text{StateOn}(\text{stateA}), \\
\quad \text{Contingency}(\text{currentState} + \text{Condition}(x > 10)))) \land \\
\text{Proven}(\text{Occurs}(\text{StateOn}(\text{stateA}), \\
\quad \text{Contingency}(\text{currentState} + \text{Condition}(x \leq 10))))
\]

then from AX22 (Occurrence proven for two state descriptors is proven for union) it is proven that:

\[
\text{Proven}(\text{Occurs}(\text{StateOn}(\text{stateA}), \text{Contingency}(\text{currentState}))).
\]

Hence it would be possible for the contingency inferences to produce circular assertions (which invalidates the logical retraction mechanism). There is a natural ordering of the contingency descriptions — e.g.

\[
\text{Contingency}(\text{currentState})
\]

is ‘higher up’ than

\[
\text{Contingency}(\text{currentState} + \text{Condition}(x > 10))
\]

since it is a superset. This ordering is used to mark Proven facts as having been proven using ‘up’ or ‘down’ inferences. The inference rule for AX21 (Occurrence proven for a state descriptor is proven for more specific state) will only fire if:

\[
\text{Proven}(\text{Occurs}(\text{StateOn}(\text{stateA}), \text{Contingency}(\text{currentState})))
\]

is not marked with direction=‘up’ and it asserts:

\[
\text{Proven}(\text{Occurs}(\text{StateOn}(\text{stateA}), \\
\quad \text{Contingency}(\text{currentState} + \text{Condition}(x > 10)))) \text{ with a direction of ‘down’}.
\]

The rule AX22 (Occurrence proven for two state descriptors is proven for union) will
only fire if neither the

\[ \text{Proven}( \text{Occurs}(\text{StateOn}(\text{stateA}), \text{Contingency}(\text{currentState} + \text{Condition}(x > 10)))) \]

or \[ \text{Proven}( \text{Occurs}(\text{StateOn}(\text{stateA}), \text{Contingency}(\text{currentState} + \text{Condition}(x \leq 10)))) \]

is marked with direction = ‘down’ and asserts:

\[ \text{Proven}( \text{Occurs}(\text{StateOn}(\text{stateA}), \text{Contingency}(\text{currentState}))) \]

with a direction of up. Including this consideration of direction in the rules avoids the possibility of circular assertions.

**Reducing redundant deductions about occurrences**

As an optimisation to minimise redundant inferences about occurrences on contingencies, the planner only reasons using the most general proven occurrence with respect to the contingencies on which the occurrence is proven. The planner ensures that the most general form of a proven occurrence is reasoned with by marking the more specific versions of the proof to be ignored by the planner. E.g. if:

\[ \text{Proven}( \text{Occurs}(\text{StateConditionOn}(x = \text{true}, \text{stateA}), \text{Contingency}(\text{currentState}, + \text{Condition}(y = 5) + \text{Condition}(z = 10)))) \]

and

\[ \text{Proven}( \text{Occurs}(\text{StateConditionOn}(x = \text{true}, \text{stateA}), \text{Contingency}(\text{currentState}, + \text{Condition}(z = 10)))) \]

Then the planner asserts the metapredicate:

\[ \text{ProofSubsumed}( \text{Occurs}(\text{StateConditionOn}(x = \text{true}, \text{stateA}), \text{Contingency}(\text{currentState}, + \text{Condition}(y = 5) + \text{Condition}(z = 10)))) \]
This predicate asserts that the occurrence is already proven for a superset set of contingencies (in this case the occurrence is proven on the superset of contingencies defined by \texttt{currentState, +Condition(z = 10)}. Since the directed deduction rules specify that any proofs in the antecedent are not subsumed, no deductions would occur off the basis of this subsumed fact. All further deductions will be made based on the more general fact:

\[
\begin{align*}
\text{Occurs}(&\text{StateConditionOn}(x = \text{true}, \text{stateA}), \\
\text{Contingency}(&\text{currentState, +Condition(z = 10)}))
\end{align*}
\]

**Workaround for limitations of drools logical retraction mechanism**

**Logical retraction** When a fact is inserted into the rules engine working memory, if it asserted using the logical mechanism, then that fact will be retracted if any of the antecedents of the rule are no longer true. Abductive rules only assert subgoals or facts about the initial plan state, neither of which should be retracted during plan extension, so the logical assertion is never used in the abductive rules — only non logical assertions are made (which will never be retracted even if the rule antecedents are no longer true). Additionally the metapredicate \texttt{ProofInProgress} is also asserted non logically, otherwise the rule would cause it own retraction, since the absence of any \texttt{ProofInProgress} for the subgoal is part of the rule antecedent.

For directed deduction rules which perform forward inferencing from the initial plan and world state any \texttt{Proven} assertions made by these rules must be made logically so the proven state, protection occurrences are consistent with the most current plan state. As discussed above, by addition of new agent actions or new action start conditions some of the previously proven state, protection occurrences may no longer be true. Directed rules include the metapredicates \texttt{NeedsProof} or \texttt{Disproof} in their antecedents since these rules should only fire when they are addressing a subgoal. As discussed in the previous section, during plan extension existing subgoals may be ‘re-scoped’ to a more restricted contingency (this consists of retracting the original subgoal and creating the new restricted subgoal in its place). The deductive rules should only be retracted if one of the \texttt{Proven} or \texttt{Disproven} antecedents is retracted. The rule should not retract if the subgoals which form part of the rule antecedents are retracted — as these do not impact the validity of the assertions the
rule has made. The drools logical retraction mechanism does not distinguish between the
different forms of antecedent for establishing when a rule fire needs to be retracted. The
workaround for avoiding retraction based on a retraction of the subgoal was achieved by
introducing a new metapredicate — *Piton*, which contains a description of the subgoal being
addressed and the name of the rule which asserted proof of the subgoal. The antecedent
for the rule is defined as:

$$(\text{NeedsProof}(\text{predicate}) \text{ OR } \text{Piton}(\text{predicate})) + \text{ whatever proven facts form the rest of the antecedents.}$$

When the rule fires, it asserts the *Piton*. The presence of the Piton prevents retraction
of the rule fire even if the subgoal *NeedsProof* is subsequently retracted.

**Prevention of multiple rule fires**

Drools sometimes fires the same rule for the same antecedents — in order to prevent this
wasteful re-firing additional code was written to store a record of all fired rules and to avoid
performing the rule body if the same rule and antecedent set has fired before.

**5.1.8 Example rules**

**Example deductive rule**

An example of a forwards inference rule is shown below. This rule proves occurrence of
a condition on state B from occurrence of a predecessor supporting state and protection
of that support. The *Piton* predicate is used to prevent the retraction of the rule if the
*NeedsProof* predicate is retracted — since as previously discussed we do not wish to retract
valid inferences during the subgoal re-scoping that occurs when determining remaining
unproven contingencies. The *Piton* is also used to avoid retraction in the case that any
of the proofs become subsumed since the subsumption of a proven occurrence does not
invalidate any previous proofs which may have been derived from the subsumed occurrence
proof.

(Note use of the double negative check ” not (not NeedsProof(...)” was necessitated by a
drools bug with the use of the or boolean operator).
rule "Deduction: State A condition directly implies state B condition and stateA occurs and protection from stateA to B implies that B occurs on trajectory of A"

salience (DEDUCTION_SALIENCE)

when

// Needed occurrence.

$neededConditionOccurrence : Occurs()

not (not NeedsProof(predicate==$neededConditionOccurrence)
     and not Piton(fireName == ("supportAndProtection" +
      $neededConditionOccurrence.getDescriptor())))

$neededConditionState : StateConditionOn(
   this == $neededConditionOccurrence.trajectoryPredicateDefinition)

// Implied by some other state condition.

$implies: ConditionImplies($supportingCondition: conditionA,
   conditionB == $neededConditionState.condition, binding1==null,
   binding2 == null, requiredAssumption == null )

Proven(predicate == $implies)

// Supporting state Occurrence is proven on some contingency.

$holdsSupporting : Holds(statePredicateDefinition==$supportingCondition)

Proven(predicate == $holdsSupporting )

$supportingState: State(this == $holdsSupporting.state)

$supportingStateOn : StateOn(state == $holdsSupporting.state )

$supportingStateOccurrence : Occurs(trajectoryPredicateDefinition ==
   $supportingStateOn )

$proven: Proven(predicate == $supportingStateOccurrence)

not (ProofSubsumed(proofOrDisproof == $proven)
     and (not Piton(fireName == $proven.descriptor))))

$supportingContingency : Contingency(this ==
   $supportingStateOccurrence.trajectory)
// Make sure state A is either an initial state or effect state
// (ie NOT a trigger state).
not TriggeredEvent(triggerState == $supportingState)

// Protection between supporting state and the needed state is proven
// on some contingency.
$protection: ProtectionOn(supportingState == $supportingState,
    supportedState==$neededConditionState.state,
    protectedCondition == $neededConditionState.condition)
$protectionOccurrence : Occurs(trajectoryPredicateDefinition == $protection )
$protectionContingency : Contingency(this == $protectionOccurrence.trajectory )
$protectionProven: Proven(predicate == $protectionOccurrence)

// Ordering on between supporting state and the needed state is proven on
// some contingency.
$ordering: OrderingOn(fromState == $supportingState,
    toState==$neededConditionState.state)
$orderingOccurrence : Occurs(trajectoryPredicateDefinition == $ordering )
$orderingContingency : Contingency(this == $orderingOccurrence.trajectory )
$orderingProven: Proven(predicate == $orderingOccurrence )
not (ProofSubsumed(proofOrDisproof == $orderingProven)and
    (not Piton(fireName == $proven.descriptor)))

then

// Proof of needed state condition occurrence is on the intersection of
// supporting state and the protection trajectory.
Planner.printRuleInfo(drools );
// Contingency is named for the occurrence.
String contingencyStateName = "cont_" + $supportingState.getDescriptor() + 
    "_" + $protection.getDescriptor() + ".state";
Contingency supportedContingency = Contingency.createAndInsertIntersect(drools,
    $supportingContingency, $protectionContingency );

// Ordering also needs to be proven
Contingency provenContingency = Contingency.createAndInsertIntersect(drools,
    supportedContingency, $orderingContingency );
Occurs occurs = (Occurs) Planner.insertLogical(drools,
    new Occurs($neededConditionState, provenContingency));
Planner.insertLogical(drools, new Proven(occurs));
Planner.insertLogical(drools, new Piton("supportAndProtection" +
    $neededConditionOccurrence.getDescriptor()));
end

The body of the rule calculates the proven contingency for the condition occurrence in state B as the result of intersecting the proven contingencies for the supporting state A occurrence, the proven contingencies for occurrence of ordering from A to B and the proven contingencies for protection of the condition from state A to state B.

Example abductive rules

Abduction: agent action contingency control by contingency decision fluent"

This abductive rule adds contingency control to an agent action by predicating the agent action upon a planner variable. It fires when there is an agent action occurrence which needs to be proven on one contingency and which needs to be disproven on another contingency (this subgoal can arise via a need to protect a threat this action poses to other actions).

rule "Abduction: agent action contingency control by contingency decision fluent"
no-loop true
// High salience since this is the ‘best’ way to control an action application..
salience (ABDUCTIVE_SALIENCE+200)
when
// Needed agent action occurrence.
$neededActionOccurrence: Occurs()
$needsProof: NeedsProof(predicate == $neededActionOccurrence)
$neededStateOn: StateOn(this == $neededActionOccurrence.trajectoryPredicateDefinition)
// The needed state is a trigger state for an agent action.
$agentActionTriggeredEvent: TriggeredEvent(
    triggerState == $neededStateOn.state)
TriggeredEventDefn(this == $agentActionTriggeredEvent.eventDefn,
    eventType == TriggeredEventType.AGENTSTARTACTION)
The same event needs to be disproved on some other contingency.

$\text{notneededActionOccurrence} : \text{Occurs}(\text{trajectoryPredicateDefinition} == \text{$\text{neededStateOn}$})

$\text{needsDisProof} : \text{NeedsDisProof(predicate} == \text{$\text{notneededActionOccurrence}$})

// Use eval to prevent logical retraction.

$\text{not DisProofInProgress(predicate} == \text{$\text{notneededActionOccurrence}$})$

then

Planner.printRuleInfo(drools);

if (!Planner.checkForPreviousActivation(drools))
{
    Planner.setProofInProgress(drools, $\text{needsProof}$,
        "$\text{addNodeContingencyControl}$");
    Planner.setDisProofInProgress(drools, $\text{needsDisProof}$,
        "$\text{addNodeContingencyControl}$");
    // Add in contingency control for this node.
    Planner.addNodeContingencyControl(drools,$\text{agentActionTriggeredEvent}$,
        (Contingency)$\text{neededActionOccurrence}.\text{getTrajectory}$(),
        (Contingency)$\text{notneededActionOccurrence}.\text{getTrajectory}$());
}
end

The java method $\text{addNodeContingencyControl}$ creates a control planner variable (with a name which describes the contingencies where the agent action is needed. It adds truth of the control variable to the trigger state for the agent action start event. It specifies that the initial value of the control planner variable is "$null$". The planner then sets up two subgoals — one to prove that condition that the control variable is true occurs in the action start state on contingencies where the action is needed and the other to disprove that the condition that the control variable is true occurs in the action state state on those contingencies where the agent action needs to be disproved.

Abduction: try to prove INDIRECT condition support from an existing (proven) state in plan

This abductive rule seeks support for a needed state condition occurrence by creating subgoals to prove occurrence of a new event and a required assumption condition together
which provide support for the needed state condition. (For example it can provide support for the condition \( x = y \) from an event which achieves \( x = 5 \) and the required additional condition that \( y = 5 \)). The head of the rule confirms that the required assumption is achievable — either because there are initial contingencies in which it holds, or because there are events which can bring about the required assumption condition.

```plaintext
rule "Abduction: Prove condition support by assuming a new event
(agent action or not) which INDIRECTLY causes the needed condition"
saliency (ABDUCTIVE_SALIENCE)
when
  // Needed state condition occurrence.
  $neededConditionOccurrence : Occurs($trajectory: trajectory,
   $stateConditionOn : trajectoryPredicateDefinition )
  $neededContingency : Contingency(this ==
   $neededConditionOccurrence.trajectory )
  $neededStateCondition : StateConditionOn(this ==
   $neededConditionOccurrence.trajectoryPredicateDefinition )
  $needsProof : NeedsProof(predicate == $neededConditionOccurrence )
  // Event type causes the needed condition with required assumptions.
  $eventTypeCauses: EventTypeCauses(causedCondition==
   $neededStateCondition.condition)
  // Exclude agent node start events --- these events are explicitly instantiated
  // when the exec event is instantiated.
  $eventDefn : TriggeredEventDefn(this == $eventTypeCauses.eventDefinition,
   eventType != TriggeredEventType.AGENTSTARTACTION )
  $requiredAssumption : Condition(this==$eventTypeCauses.requiredAssumption )
  // Make sure the requiredAssumption is achievable DIRECTLY (ie no unviable )
  not ( |
    ($holdsInCurrentState : Holds(state == Planner.currentState ||
      state == $neededContingency.startState)and
    NegativeConditionImplies(conditionA ==
      $holdsInCurrentState.statePredicateDefinition,
      conditionB == $requiredAssumption)
and
not EventTypeCauses(causedCondition==$requiredAssumption,
requiredAssumption == null)
)
not ProofInProgress(predicate == $neededConditionOccurrence )
then
Planner.printRuleInfo(drools );
if (!Planner.checkForPreviousActivation(drools))
{
// Assert triggered event occurrence with the bindings needed to match
// the needed condition.
Planner.setProofInProgress(drools, $needsProof,
($eventTypeCauses.getDescriptor() + "+
+ $requiredAssumption.getDescriptor());
TriggeredEvent triggeredEvent = Planner.createSupportFromNewEvent(drools,
$eventTypeCauses,
$neededConditionOccurrence );
// We also need to try to prove that the required assumed condition occurs
// in the supported state.
StateConditionOn assumptionStateConditionOn =
(StateConditionOn) Planner.insertLogical(drools,
new StateConditionOn($requiredAssumption, $neededStateCondition.getState()));
Occurs assumptionOccurs = (Occurs) Planner.insertLogical(drools,
new Occurs(assumptionStateConditionOn,
$neededConditionOccurrence.getTrajectory()));
Planner.insertNeedsProof(drools, new NeedsProof(assumptionOccurs));
end

The body of the rule creates a description of the needed supporting event and calls the
java method createSupportFromNewEvent which creates a subgoal to prove prior occurrence
of that event and to prove protection of that supporting condition from that supporting
event to the needed state. It also creates a subgoal to prove occurrence of the assumption
condition.
For reasons of brevity, only a couple of key rules are shown here, the names of the complete list of rules used by the planner is provided in the appendices.
Chapter 6
CONCLUSIONS AND FUTURE WORK

This chapter summarises the contributions, discusses the strengths and weaknesses of the approach and describes future extensions which could be made to the planner.

6.1 Conclusions

6.1.1 Contributions

Based on a real-world domain analysis this thesis identifies the set of features required of a planner for the domain of computer batch job scheduling namely:

- Reasoning with states involving numerical fluents and conditions using $<$ and $>$ operators.
- Handling of triggered events.
- Ability to plan with simple time goals.
- Planning with events of variable duration and creation of plans which perform action monitoring.
- Compact representation of contingent plans.
- Planning for knowledge acquisition for both propositional and numerical values and subsequent use of that knowledge.
- Allowance of action execution without verified preconditions.

From a review of existing contingent planners none individually offer all of the required features. This dissertation presents a new flexible planner to address these required features.
in a fully integrated manner. The representation models the plan as a dynamic system which gives the plan a very clear semantics. The plan representation draws upon the existing batch job scheduler representations whose viability as an execution model has already been demonstrated in the real-world. The dissertation presents a partial ordered triggered event action logic for reasoning about this dynamic system and describes the implementation and evaluation of a prototype planner \textit{JobPlan} which is able to generate plans for key examples from the domain using abductive reasoning with this action logic.

Modeling the plan as a dynamic system allows planning aspects such as action monitoring, contingent planning, knowledge acquisition, knowledge use, and temporal planning use the same logic to be reasoned about in an integrated way — allowing the interaction between knowledge acquisition, action execution and contingent planning to be modeled and handled. However, by representing the plan in this way, all effects of the plan must be established using dynamical inferences — entailing a high number of inferences.

Although created for the computer batch job domain this representation can be applied in any domain where actions are triggered in response to external events — this includes workflows and event driven systems.

### 6.1.2 Drawbacks of the approach

The planner offers a representation and reasoning which addresses a broad set of requirements — however the generality of the representation incurs a computational price and a drawbacks of this approach is that the generation involves a high number of inferences. Every action requires a minimum of two events to describe it. Explicit backwards and forwards inferences are used which doubles the number of inferences versus a pure progression or regression approach. The causal support for every trigger condition must be explicitly proven which entails proving the protection and ordering for the causal support. The successful proof of repair DB example problem (which has 7 planner actions) consists of 1402 forwards inferences and 588 abductive inferences.

The breakdown of the forward inferences is as follows:

Note the number of facts regarding proven state occurrences is higher than the number of event states (which is 31 for the repairDB example). This is because during plan
Table 6.1: Type and number of inferences in the proof of plan correctness for the repairDB example

<table>
<thead>
<tr>
<th>Inference</th>
<th>Number of inference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proven Ordering</td>
<td>464</td>
</tr>
<tr>
<td>Proven Protection</td>
<td>97</td>
</tr>
<tr>
<td>Disproven threats</td>
<td>87</td>
</tr>
<tr>
<td>Proven StateCondition</td>
<td>315</td>
</tr>
<tr>
<td>Proven StateOn</td>
<td>131</td>
</tr>
<tr>
<td>DisProven StateCondition</td>
<td>164</td>
</tr>
<tr>
<td>DisProven StateOn</td>
<td>144</td>
</tr>
</tbody>
</table>

construction the occurrence of states must be proven on each different contingency (in this case 3).

6.2 Future Investigations

6.2.1 Using the plan variable representation with a forwards based planner

In domains where the effect of all actions is immediate (i.e. duration = 1 time unit) and exogenous events do not occur, the use of a partial order action logic supporting triggered events is not needed. A simple immediate deterministic transition function may be used to determine the resultant state given the current state and the agent action being applied. With these simplifications the use of the plan variable knowledge representation could be used with a forwards search approach such as used by contingent-FF (Hoffmann, 2005). With use of the plan variable representation, the belief state consists of the value mapping for the automatic fluents and the plan variables. The plan consists of a tree where each node is an action and where the decision on which branch in the plan is to executed is decided by the values held by either the automatic or the plan variables. During plan construction the planner may augment the plan and the current belief state via one of the following changes:

- Creation of a new plan variable. When the current belief state mapping is augmented with a mapping for the variable value and all future belief states must include the
mapping for this plan variable.

- Addition of an action (which may assign an output result to a plan variable) to a leaf node. The successor belief state for the branch is obtained by applying the action to the current belief state.

- Addition of a new branch with the branch predicated on the value of one or more automatic fluents or plan variables.

6.2.2 Possible extensions of the planner

The design of the planner is highly extensible. Its use of abductive inferences to generate the plan has the attractive property whereby new abductive rules may be added without impacting planner correctness or completeness. New high level abductive rules could be added to encode high level plan design approaches such as the insertion of ‘pre-built’ sub-plans (for example a sub-plan which implements an efficient diagnostic approach) into the overall plan. Such plans would then be automatically validated with the existing forwards dynamical inferences. Hypothetically the following extensions could be made within the existing planner architecture and building these out would be an interesting avenue of further research.

Temporal reasoning

Due to time constraints and the complexity of the implementation the temporal reasoning abilities of the prototype planner have not been evaluated. However from the described proof outline from the evaluation chapter, the manner in which the representation could handle contingent temporal reasoning has been presented.

Contingent plan with re-planning, memory and diagnosis

In general it is preferable to plan for all contingencies at plan generation time. This allows the full behaviour of the plan to be made known to human operators and reviewable prior to execution. Additionally since the same plan may be run multiple times (for example daily) then less computation is required if the plan can be generated for all contingencies apriori.
However, if the problem domain is sufficiently complicated it may not be worth the planner constructing a plan to cover those contingencies which are deemed to be very unlikely. The planning representation could support re-planning by defining a *REPLAN* action which the planner can include as an action in the plan. If the plan execution activates this action, it then performs a re-plan. Note this means that the planner behaviour is committed in its behaviour up until it executes a re-plan action, hence the representation gives a clear definition of plan commitment as per the requirements outlined by Pollack (1992). If during plan execution a re-plan activation is reached the planner invokes its deliberative planning algorithm to generate a plan based on the current situation. The existing state of the plan is not modified, instead additional actions are added to handle the current situation (which was not previously accounted for in the plan). These actions are ordered after the re-plan action. The planner may only add components to the plan. This means that actions which are predecessors to the re-planning action remain unaffected and hence these may form the basis for the planner’s record of past actions and fluent values.

In the example plan below, if the report generation job is successful it then runs the ftp job, but if the report generation fails and the REPLAN command is activated, the planner performs deliberative planning to handle the exception situation.

```
(name: i_reportStatus, value: null )
(name: genReport1220,
  command:"i_reportStatus = runReport 1220" ,
  status:Initialised)
(name:ftpToRemote_Report1220,
  command: "ftp Report1220",
  status:Initialised,
  startConditions: Report1220.exists=True,
  genReport1220.status = Complete,
  i_reportStatus=0)
(name:replan,
  command: REPLAN,
  status:Initialised,
```
startConditions:
    genReport1220.status = Complete,
    i_reportStatus<>0)

Record of past events

In a fully observable world which is a Markov process (i.e. future behaviour is defined completely by the current state) there is no need for the agent to have a memory of past events since all information relevant to future evolution is encoded in the current state of the system. In a partially observable environment, the agent’s beliefs about the external world may be incomplete or inaccurate. As demonstrated in this example beliefs about past events may be required in order to make the correct inferences about the current state of fluents which are not directly observable. For example in the circuit diagnosis example in Halpern (2005), the agent is required to remember all past observations about the circuit which are stored as a completely ordered sequence of epistemic fluent values.

The plan representation may serve as a simple memory which suffices for diagnosis as follows:

- From an action job in completed status the planner can assert that an event of the type defined by the action took place.

- Any planner variables which were assigned the results of sensing actions — define the value of the corresponding external value fluent at the time the sense action was executed.

In the illustrated example the agent the plan contains a replan action which is executed under the circumstance that the report job run completes with a non zero exit status. If the replan node is executed the planner invokes deliberative planning inferences to establish plan changes to bring about the goal (existence of the report).

When the planner re-plans for the goal it must be able to reason that it is in a contingency where one or other of the preconditions for a successful report run were not met. (If it weren’t able to do this it might simply run the job again and end up in an infinite execute — replan — execute loop which would continually produce a failed result.)
The planner may establish that one of the preconditions does not hold by making the following inferences from the current state of the plan:

- The run report job action was executed in the past (abductive backwards temporal inference from current planner state where report job status \( \text{genReport1220.status} = \text{Completed} \) — the only event that caused a job status to be completed is execution of that job)

- The run report job action was unsuccessful (abductive backwards temporal inference from current planner state \( i._\text{reportStatus} <> 0 \) — the only event which causes \( i._\text{reportStatus} <> 0 \) is the report job unsuccessful execution).

- At the time of executing the run action one or more of the preconditions for a successful job run event was not true. (From abductive reasoning that the only event which would have led to an unsuccessful run is if the action was started when one of the preconditions didn’t hold).

Once the planner has established that one or more of the preconditions was not met, it then subgoals to repair the broken precondition by inserting appropriate repair actions with knowledge subgoals as needed. Note that during this re-planning the planner is effectively performing a form of belief revision since the original plan was based on the implicit assumption that the preconditions for the report generation were met, however during re-plan the planner revises this to now plan on the basis that one of the preconditions for the action is not met.

**Rule learning**

While not explored in this dissertation, an interesting extension would be to use rule learning techniques to have the planner learn more specific and targeted inference rules based on example problems. The technique of chunking (Laird, Rosenbloom, & Newell, 1986) detects common firing patterns for rules and creates a single rule which does the work of a sequence of smaller rule firings. This allows commonly used rule firings to be performed more quickly. Such an approach could lead to higher level abductive and deductive causal rules such as
those used by humans for planning. This would improve efficiency and shorten the search for proofs (via the chunking of abductive rules) and the proofs themselves (via the chunking of deductive rules).
Appendix A

Case Study Scenarios

This appendix documents the scenarios from the case study. The description of each scenario includes the problem observed during job execution, the corrective actions taken by the support staff and the forms of planning activities required by the support staff for the scenario.
<table>
<thead>
<tr>
<th>Title</th>
<th>Observed Problem</th>
<th>Root Cause</th>
<th>Corrective Action</th>
<th>Required Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource requirements</td>
<td>Data input load fails</td>
<td>Lack of disk space</td>
<td>Clear up disk space. Rerun input load script</td>
<td>Numerical condition effects, Contingent planning, Action execution with unverified preconditions</td>
</tr>
<tr>
<td>Event monitoring</td>
<td>Send payments file, acknowledgment not received in nominal response time</td>
<td>External payments system down</td>
<td>Notify external team</td>
<td>Event monitoring, Exogenous events, Durative events</td>
</tr>
<tr>
<td>Job overrun</td>
<td>DB Dump process does not complete by expected end time</td>
<td>DB internal error</td>
<td>Alert DB team.</td>
<td>Temporal effects, Exogenous event</td>
</tr>
<tr>
<td>Concurrent actions</td>
<td>run 3 jobs to generate 3 required report types. Balance report fails</td>
<td>DB blocking</td>
<td>Examine log file to determine cause of failure. Unblock DB, rerun balance report then generate overall marker file</td>
<td>Concurrent action monitoring, Diagnosis using external record</td>
</tr>
<tr>
<td>DB error diagnosis</td>
<td>DB start job fails</td>
<td>DB index error</td>
<td>Run multiple db diagnostics to determine error. Run DB repair job for appropriate error, restart DB</td>
<td>Diagnosis using multiple sense actions.</td>
</tr>
<tr>
<td>Title</td>
<td>Observed Problem</td>
<td>Root Cause</td>
<td>Corrective Action</td>
<td>Key Required Capability</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------------------------------------------------------------------</td>
<td>---------------------</td>
<td>----------------------------------------------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>Daemon monitoring</td>
<td>Daemon process stops before scheduled end time</td>
<td>Disk full</td>
<td>Clear disk space, Restart the daemon.</td>
<td>Non nominal event durations, Diagnosis of failed precondition</td>
</tr>
<tr>
<td>External dependencies</td>
<td>Output feed (client process) fails</td>
<td>Application server went down.</td>
<td>Restart the application server, check it is running okay. Re run the output feed</td>
<td>Diagnosis</td>
</tr>
<tr>
<td>Bad postconditions</td>
<td>Trade output feed (client process) runs, but generates an output file which only contains header information and no data, file size less than required.</td>
<td>The messaging layer between the server and the client had a temporary performance issue.</td>
<td>Check that the server is running okay. Then re-run the output feed</td>
<td>Detection of non nominal event effects.</td>
</tr>
<tr>
<td>Restoration of preconditions</td>
<td>Input load (client process) of a trade input file times out and completes in time ≠ nominal duration</td>
<td>Application server on which client depends has degraded performance</td>
<td>Restart the application server. Move input cash flow file back to the input directory. Notify ETA to downstream system support team</td>
<td>Non nominal duration</td>
</tr>
</tbody>
</table>
Table A.3: Problem Scenarios for Example System — continued

| Title              | Observed Problem                                                                 | Root Cause                                                                 | Corrective Action                                                                 | Key Required Capability                                     |
|--------------------|----------------------------------------------------------------------------------|                                                                           |                                                                                     |                                                        |
| Schedule conflict  | Multiple client process failure, database performance=Slow                       | Database maintenance job was mis-scheduled                                | Stop the maintenance job, restart application server. Rerun all failed client processes, run maintenance job at later time | Shared resource reasoning, threat prevent via time constraints. |
| Diagnosis using log file | Input load (client process) of a cash flow input file fails                        | database login expired.                                                   | One of the preconditions has not been met. Parse log file for error to determine that database login state is has expired state. Un-expire the password. Re-re-run the input load program | Sensing using external record.                           |
| Prioritization     | Two reports report jobs fail.                                                     | CPU usage too high, client time out.                                     | Start the job with the closer deadline. Let it run to completion before the lower priority job is started. | Time based goals, Resource requirements                  |
Table A.4: Problem Scenarios for Example System — continued

<table>
<thead>
<tr>
<th>Title</th>
<th>Observed Problem</th>
<th>Root Cause</th>
<th>Corrective Action</th>
<th>Key Required Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support alert management</td>
<td>Connection to message queue fails</td>
<td>message channel process is down</td>
<td>Contact support group for message queues. Wait for them to acknowledge rectification of the problem</td>
<td>Triggering of exogenous event and monitoring.</td>
</tr>
<tr>
<td>Function of time</td>
<td>Ledger Report failed</td>
<td>Ledger data date is not = current date because job was temporarily disabled for 2 runs</td>
<td>Run the ledger job 2 times</td>
<td>Function based preconditions</td>
</tr>
<tr>
<td>Delay diagnosis</td>
<td>Payment in database is stuck at non completed status</td>
<td>Blockage in inbound message queue</td>
<td>Perform tests for each possible cause. Determine cause, run repair action. Restart the inbound message processor.</td>
<td>Indirect sensing</td>
</tr>
<tr>
<td>Data precondition</td>
<td>Report failed</td>
<td>Rate database did not contain data for the required date</td>
<td>Contact rates database support group and inform them that the rates information is non existent for that date job.</td>
<td>Functional precondition, Exogenous event</td>
</tr>
<tr>
<td>Deadlines</td>
<td>Input of external trade feed data fails, drop dead time of trade import passed.</td>
<td>External system did not provide the file by the specified time.</td>
<td>Notify external system support. Once the file has been received, re-run the trade import job.</td>
<td>Non nominal duration, Exogenous event</td>
</tr>
<tr>
<td>Title</td>
<td>Observed Problem</td>
<td>Root Cause</td>
<td>Corrective Action</td>
<td>Key Required Capability</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Performance monitoring</td>
<td>Clients and server performance is slow</td>
<td>Messaging layer has deadlocked.</td>
<td>Run a undeadlock command. Apps server performance returns to normal. Clients do not need to be restarted.</td>
<td>Performance monitoring and inference of exogenous deadlock event.</td>
</tr>
<tr>
<td>Job overrun</td>
<td>Database load job runtime &gt; expected maximum duration</td>
<td>Database load did not complete due to bug in database software.</td>
<td>No automated correction known. Contact database support team.</td>
<td>Non nominal duration, Exogenous event</td>
</tr>
<tr>
<td>Functional preconditions</td>
<td>Database dump fails due to lack of disk space</td>
<td>Required disk space for database dump is function of the database size</td>
<td>Determine the required space, clear that much space.</td>
<td>Functional precondition</td>
</tr>
<tr>
<td>Conditional effects</td>
<td>Batch job fails which processes all files in an input directory fails.</td>
<td>Database slowdown</td>
<td>Move files which were in the input directory back to the input directory and reprocess.</td>
<td>Conditional effects</td>
</tr>
<tr>
<td>Ramifications</td>
<td>Db server crashes</td>
<td>Internal server issue</td>
<td>Database server must be restarted. Restart all running processes which are accessing the databases</td>
<td>Ramifications</td>
</tr>
</tbody>
</table>
Appendix B

Generated plans

This appendix provides the generated plans for each implemented example. For brevity the full search and generated plan proofs are not included here.

B.0.3 Example plan 1 — Generate and ftp report

Handling exogenous events, durative action monitoring and triggered actions — A report script generates a report for a given date by processing an input file. The input file is only received after it has been generated by an exogenous external event. The report generation takes a variable amount of time and must be monitored for completion. Once the report is generated the report file is ftped to a remote server for use by an external job. This examples demonstrates planning with exogenous events, triggered events and action execution and monitoring of durative actions. (Note the naming conventions used in the implemented planner differ slightly from the notation used in the description of the planner.)

name: report_node_1,
command: "runReport.sh 20091220",
start condition: Condition((File)externalInputFile.exists,==,True)

name: ftpAFileToRemoteServer_node_0
command: "ftp.pl Report20091220"
start condition:

B.0.4 Example plan 2 — determine error and repair DB

Planning with knowledge goals and knowledge use with merged plan branches — A database error must be repaired, where the error value can be 1, 2 or 3. To check for internal database
errors a test script *checkDb* *e* can be run which takes as an argument the error condition *e* it is checking for and outputs *true* if the database has that error or *false* if it doesn’t. There is also a *repairDB* script which takes as an argument an error number *e* and repairs that error condition (or which has no effect if the database does not have that condition). Using these scripts, if a database has an error, the error condition may be determined using the *checkDb* script and once the error number is determined the *repairDB* script can be called with this error number to remove that error condition and return the database state to nominal.

name: checkDB2_node_16
command: "i_control_14.Condition(db.state,,2) = test 2"
start condition:

name: checkDB1_node_21
command: "i_control_19.Condition(db.state,,1) = test 1"
start condition:

name: checkDB3_node_11
command: "i_control_9.Condition(db.state,,3) = test 3"
start condition:

name: assign2_node_13
command: "i_db.state = 2"
start condition: Condition(checkDB2_node_16.status,==,Completed) AND Condition(i_control_14.Condition(db.state,,2),==,True)

name: assign1_node_18
command: "i_db.state = 1"
start condition: Condition(checkDB1_node_21.status,==,Completed) AND Condition(i_control_19.Condition(db.state,,1),==,True)

name: assign3_node_8
command: "i_db.state = 3"
start condition: Condition(checkDB3_node_11.status,==,Completed) AND
Condition(i_control_9.Condition(db.state,,3),==,True)

name: repairDBError_6_node_7
command: "repairDB i_dbState"
start condition: Condition(assign1_node_18.status|
assign2_node_13.status|
assign3_node_8.status,==,Completed)

B.0.5 Example plan 3 — Sensing using exogenous event and external recording medium

Sensing using exogenous action to obtain knowledge and use of an external medium to record knowledge — a variant of the above example where a database has an internal error, but in this example the value of the database error is determined by an exogenous event and communicated to the planner via an external medium (email).

name: sendRequestToDBA_node_5
command: "sendRequestToDBA"
start condition:

name: readInboxValue_node_4
command: "i_db.status = PlannerInbox.value"
start condition: Condition(PlannerInbox.hasMessage,==,true)

name: repairDB_node_3
command: "repairDB i_dbState"
start condition: Condition(readInboxValue_node_4.status,==,Completed)
Appendix C

Planner Rules

This appendix provides the list of the names of rules used in JobPlan.

1. “Abduction: agent action contingency control by contingency decision fluent”

2. “Abduction: Proof of protection requires disproof of every potential threat”

3. “Abduction: Prove condition support by assuming a new event (agent action or not) which INDIRECTLY causes the needed condition”

4. “Abduction: Prove contingent internal belief assignment by a new propositional sensing action which provides the needed contingent assignment”

5. “Abduction: Prove required ordering between agent actions using explicit node dependency”

6. “Abduction: Provide condition support by assuming a new event (agent action or not) which directly causes the needed condition”

7. “Abduction: To disprove a state CONDITION occurrence on a contingency INDIRECTLY causing events, disprove the required assumption”

8. “Abduction: To disprove a state CONDITION occurrence on a contingency need to disprove any existing events which cause it”

9. “Abduction: To disprove a state CONDITION occurrence on a contingency, need disproof on all potentially directly causing events”

10. “Abduction: To disprove an agent exec action on a contingency, try disproof of the status=Executing condition”
11. “Abduction: To disprove an effect state occurrence on a contingency, need to prove that trigger doesn’t happen”

12. “Abduction: To disprove an exogenous event trigger state occurrence on a contingency, try disproof of stateConditionOccurrence for a condition which holds”

13. “Abduction: To proven a state occurrence on a contingency, requires proof of stateConditionOccurrence for each condition which holds”

14. “Abduction: Try to prove condition support by assuming it in some contingencies of the initial state.”

15. “Abduction: leap of faith — try to prove support for a belief by specifying its initial state”

16. “Abduction: leap of unfaith — disprove support for a belief by specifying its initial state”

17. “Abduction: resolve threat via context separation”

18. “Abduction: try to prove direct condition support from an existing (proven) state in plan.”

19. “Abduction: try to prove direct condition support from an existing effect state in plan.”

20. “Abduction: try to resolve threat via threat demotion”

21. “Abduction: try to resolve threat via threat promotion”

22. “All states ordered after current or contingent state have time lower bound of current time + 1.”

23. “Can never prove an ordering prior to any initial contingency states.”

24. “DEDUCTION: If time condition EQ or UB occurrence is proven in an events trigger state and there is a duration upper bound, then bound applies to the effect”
25. “Deduction: State A condition directly implies state B condition and stateA occurs and protection from stateA to B implies that B occurs on trajectory of A”

26. “Deduction: Unique event causes a goal condition then goal is ordered after the event”

27. “Deduction: detect inconsistency”

28. “Deduction: if trigger state occurrence disproven then effect is disproven”

29. “Deduction: state occurrence on a needed trajectory disproven when state condition occurrence disproven”

30. “Detect retraction”

31. “Directed Deduction: conditions in initial contingency disprove the condition”

32. “Directed Deduction: if event is first cause of a condition then it is ordered before a trigger or goal state with that condition”

33. “Directed Deduction: protection occurrence is proven for a contingency when all threats are disproved”

34. “Directed deduction — all states occur after current state”

35. “Directed deduction — all states occur after initial contingency states”

36. “Directed deduction: occurrence disproven on a contingencyA which subsumes contingencyB is disproven on contingencyB”

37. “Directed deduction: occurrence proven on a contingencyA which subsumes contingencyB is proven on contingencyB”

38. “Directed deduction: occurrence proven/disproven on a contingencyA which subsumes contingencyB is proven/disproven on contingencyB”

39. “Directed deduction: State condition occurrence on a trajectory proven if indirect state conditions proven”

40. “Directed deduction: State condition occurrence on a trajectory proven if super condition is proved.”
41. “Directed deduction: State occurrence on a needed trajectory proven for trajectory when coincident condition occurrences are proven”

42. “Directed deduction: contingency state A does not occur on contingency B, if there is a condition in A which is negatively implied by condition in B”

43. “Directed deduction: trigger state condition occurrence disproven when it doesn’t hold in the initial contingency and all causing events triggers are disproven”

44. “Directed deduction: trigger state occurrence on a needed trajectory disproven when a super state occurrence is disproven”

45. “Directed forward inference prove ordering from transitive ordering”

46. “For a new disproof of any trajectory predicate occurrence, any needed sub-contingency disproofs are fulfilled.”

47. “For a new proof of any REQUIRED trajectory predicate occurrence determine the unproven trajectories (if any) and try to prove those”

48. “If lower time bound which holds in state B is greater than upper time bound holding in state A, then state B is ordered after state A.”

49. “If state B is proven ordered after state A on a contingency and A has a lower time bound condition then, lower time bound for state B must be > lower time bound for A”

50. “Inconsistent conditions in a state”

51. “Required proof merits a proof attempt”

52. “Store array of proven state condition occurrences”

53. “Undirected Deduction: if we have proven that threatening state does not occur on a contingency then the potential threat is disproved on that contingency.”
55. “Undirected Deduction: if we have proven that threatening state occurs after the supported state on a contingency then the potential threat is disproved on that contingency.”

56. “Undirected Deduction: if we have proven that threatening state occurs before the supporting state on a contingency then the potential threat is disproved on that contingency.”

57. “Undirected Deduction: mark contingent proof as subsumed”

58. “Undirected Deduction: If Trigger for an event is proven to occur on a contingency, then so is the effect state”

59. “Undirected Deduction: consolidate proven occurrence contingencies”

60. “Undirected Deduction: direct condition implication proved”

61. “Undirected Deduction: direct negative condition implication proved”

62. “Undirected Deduction: event trigger state is ordered before event effect state”

63. “Undirected Deduction: identify threat effect state for protection”

64. “Undirected Deduction: occurrence proven on a contingency all of whose conditions hold in the current contingency state is proven on contingency of currentContingency”

65. “Undirected Deduction: protection occurrence is DIS-proven for a contingency when any threat is proven”

66. “Undirected deduction: Left hand Fluent term matches”

67. “Undirected deduction: conditionA implies conditionB under some assumptions”

68. “Undirected deduction: event type causes needed condition”

69. “Undirected forward inferences disprove reverse ordering from ordering”

70. “check unviable conditions”

71. “ordering loop on overlapping contingencies”
72. “plan complete because all needed facts proven and threat detection completed.”

73. “plan complete subject to no threat detection — now perform threat detection and resolution”

74. “search dead-end for disproof in progress”

75. “search dead-end for proof in progress”

76. “search dead-end plan too big (> 10 nodes)”
References


Ennis, R. (1986). A continuous real-time expert system for computer operation. *IBM J. research development*, 30(0), 0.


proving to problem solving.


UC4Corp. (2002). *Uc4.* (http://www.uc4.com)


Vita

Tracey D. Lall

1987  BA in Physics Oxford University
1988  MSc in Computation Oxford University

1988-1990  Image innovation Ltd
1990-1992  SD Scicon Ltd
1993-1996  European Space Agency
1996-2010  Merrill Lynch/Bank of America

2010  “A New Representation and Planner for Computer Batch Job Scheduling, Execution Monitoring, Problem Diagnosis and Correction”, ICAART 2010