A NEW REPRESENTATION AND PLANNER FOR COMPUTER BATCH JOB SCHEDULING, EXECUTION MONITORING, PROBLEM DIAGNOSIS AND CORRECTION

Tracey Lall
Department of Computer Science, Rutgers, The State University of New Jersey, New Brunswick, New Jersey, U.S.A.

Keywords: Planning, Scheduling, Execution, Diagnosis, Automation.

Abstract: Modern enterprise computer environments use commercial schedulers to run and monitor computer batch jobs and processes. Currently the job schedules must be manually designed to include diagnosis and error correction behaviours for failed jobs or failures must be handled by support staff at execution time, requiring them to be on call while these jobs run. Automating these manual tasks using planning techniques requires a compact representation of contingent plans, handling and monitoring of actions which have a variable duration, actions which are triggered by external events and planning for knowledge goals. Currently these features are not provided by any existing single planner. We present a novel plan representation which drawing on existing scheduler representations provides all these features in an integrated manner. A planner implementation using this representation with a new action logic is described along with key worked examples from the domain.

1 INTRODUCTION

Modern enterprise computer environments involve the scheduled running of hundreds of computer batch jobs, programs and processes on a collection of machines. These consist of computer programs which are run to achieve a specific task - for example a job to generate a report and email it to a user. These jobs are executed at predetermined times and monitored for successful completion by commercial schedulers according to predefined job schedules. Schedules need to be designed in order to avoid conflicts between certain jobs (for example running a report against a database when the database is undergoing a maintenance job) and to ensure that batch job outputs are produced by the required deadlines. In all cases the job schedule definitions must be created in advance by the support team and unless explicit recovery logic is programmed into the definitions job failures will require manual intervention to diagnose the cause of the error and to take appropriate corrective actions. This makes support of such environments very costly - typically for every dollar spent on computing infrastructure, between 2-10 times that amount are spent for ongoing management (Murch, 2004).

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 error handling rules must be hand-coded by a skilled operator for each computer environment. We seek instead to create a planner which is able to generate contingent plans for job execution, monitoring and error correction based on the known behaviours of the batch jobs and the processes comprising the computer system.

From analysis of the domain there are some key aspects which need to be addressed by the planner:

- The representation of contingent plans must be in a form understandable to support staff and in order to avoid combinatorial explosion in the size of plans, the representation should be a compact representation which supports the remerging of contingent execution branches.
- The representation needs to support actions which have a variable duration and hence which require monitoring for completion.
- The representation must allow reasoning about triggered actions and events - actions and events which occur as soon as a particular set of conditions becomes true.
- The representation must support planning for knowledge goals in order to diagnose job failure root causes which are not directly sensable.

Lall T. (2010).
A NEW REPRESENTATION AND PLANNER FOR COMPUTER BATCH JOB SCHEDULING, EXECUTION MONITORING, PROBLEM DIAGNOSIS AND CORRECTION.
In *Proceedings of the 2nd International Conference on Agents and Artificial Intelligence - Artificial Intelligence*, pages 277-284
DOI: 10.5220/0002723602770284

These features are not provided by any existing single planner. We present a novel plan representation which addresses these requirements by drawing upon the existing commercial scheduler representations. The representation describes the plan as a dynamical system which in conjunction with the external world evolves/runs according to a simple dynamics. We describe an action logic for reasoning about this dynamical system which provides both forwards temporal projection inferences and means end based inferences to support partial order contingent planning. We outline the operation of an implemented planner using this logic on two key examples from the domain to demonstrate how it builds the plan in such a way that so that the combined plan and world state evolve into a goal state on all contingencies.

2 COMPUTER BATCH JOB ENVIRONMENTS

2.1 Commercial Schedulers

Automated schedulers (ComputerAssociates, 2002), (UC4, 2008) exist which allow job schedules and dependencies to be defined. A job is run once its scheduled time (if defined) is reached and its start conditions become true, e.g. the start conditions for a batch *jobC* might be:

```
success(jobA) and not running (jobB)
```

This job will run as soon as *jobA* is in *success* state (i.e. has completed with a nominal process exit status) and *jobB* is not in the *running* state. Such schedulers employ an event processor which given a set of job definitions constantly checks to see if any jobs are ready for execution.

2.2 Illustrative Example Scenarios

The following example scenarios from a case study conducted on a real world production computer environment demonstrate some of the key features of this domain:

- 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 the need for reasoning about exogenous events, triggered events and action execution and monitoring of durative actions.
- A database error must be repaired, where the error value can be 1 or 2. To check for internal database errors a test script *checkDb* 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 does nothing if the database does not have that condition). Using these scripts, in the event that a job which accesses the database fails, 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 repair the error condition.

  This example demonstrates the need for contingent planning, handling of merging of plan branches and planning for knowledge goals.

2.3 Existing Planner Applicability

There is an enormous range of existing planners with a broad range of capabilities. Any planner suitable for this domain must be able to handle the open world assumption i.e. it must be a contingent planner able to produce plans whose execution is conditional upon observations made at execution time. Planners such as Puccini (Golden, 1998), PKS (Petrick and Bacchus, 2002) , GPT (Bonet and Geffner, 2001), MBP (Bertoli et al., 2001), CC-Golog (Grosskreutz and Lakemeyer, 2000), C-Buridan (Draper et al., 1994) are all able to plan in an open world and all except for Puccini perform contingent reasoning. However these planners are unable to handle the other requirements - Puccini, PKS, MBP, GPT, MBP are all linear planners - they don't employ partial order based reasoning and hence are unable to reason about actions of varying durations. All of the planners except GPT use a branching tree plan representation where each time an observation is made and an action is predicated on that observation a branch is introduced into the plan, with no remerging of the plan branches - which can lead to combinatorial explosion in plan size. GPT uses a more compact plan representation as an MDP policy which is a mapping from planner belief states to actions.

None of the planners (except CC-Golog which has a high level $while, dowait$ language construct) include action monitoring in the plan representation.

3 NEW PLAN REPRESENTATION

A new plan representation was formulated to meet the domain requirements. The representation is a 'plan as program' type approach (as in (Levesque et al., 1997)) where the plan is a simple form of program which is interpreted at runtime into execution of actions according to the control structure and sequence defined by the program.

In order to incorporate the action monitoring aspects and to allow actions to be triggered by external conditions, the approach taken is to represent the plan itself using a very similar language as used by commercial job schedulers - where agent actions are executed as soon their associated start conditions become true. This representation's viability as a runtime execution 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 validated. This simple 'programming language' also obeys a simple formally definable dynamics and hence can be directly reasoned about using an appropriate action logic.

Most partial order planners use a mixed representation where the plan represents both plan construction time reasoning (such as causal links, threats, protections, orderings) and the runtime execution. In this representation the plan consists purely of its constituent parts described below and planner deliberation is performed seperately using an action logic which reasons *about* the dynamics of this plan.

The form for a plan in this representation is defined below:

$<plan>::=<job> \mid <planVariableValue>$
$[<plan>]$
$<job>::= (\text{"}name:\text{"} <identifier>$
$command:\text{"} <command>;$
$\text{"}status:\text{"} <status>;$
$startConditions:\text{"} <startCondition> *)$
$<status>::=\text{"}Initialised\text{"}|\text{"}Executing\text{"}|\text{"}Completed\text{"}$
$<planVariableValue>::=$
$(\text{"}name:\text{"}i\_<identifier>\ \text{"}value:\text{"} <value>)$

$<command>::= [<identifier>,=]$
$[\text{“}] <string> [<parameter> *][\text{“}]$

A command contains a string description of an internal (planner inbuilt) command or a quoted string description of an external operating system script and a list of parameters for the command. A parameter may be a plan variable or a constant. The return value from the command may be optionally assigned to a plan variable.

The command is run as soon as the job status is *Executing*. During planner deliberation for each command there are corresponding event definition(s) used by the action logic to determine the effects (on both planner variables and world fluents) of executing the command under nominal and non nominal conditions.

A *planvariable* (whose identifier is prefixed with $i\_$ to distinguish it from external world fluent identifiers) associates a value with a named variable. Its value may be set either from the results of an external command script or internal planner command (such as assignment). Such variables can be used for representing the value of fluents in the world (see below on knowledge representation). The value of a plan variable may change during execution of the plan.

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.

3.1 Plan Execution Dynamics

Execution of the plan consists of letting the plan and world state evolve according to the following dynamics, which consists of three forms of event:

- *Action Start Event* - In order to execute the plan the plan executor constantly monitors all conditions which are defined as start conditions. 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 of the conditions defined in the start conditions for a job become true and the current job state is *Initialised* the job state is set to *Executing*. If the job command is an external command it is run in the real world via the operating system command line using the specified parameters values - at which point any immediate effects of the command start in the external world take place. If the command is an internal command (such as a planner variable assignment) it executes that command and updates the specified planner variable accordingly.
- *External Action Completion* - When the action command process (which may have a varying duration) has completed (and any effects of the command taken place in the external world), any returned results from the action are assigned to the specified plan variable and the job status is set to *Completed*.

- Exogenous events - these are events which are not under direct control of the planner and occur according to various conditions becoming true in the external world. These events must also be reasoned about during plan construction.

3.2 Plan Variables and Knowledge Representation

Fluents which are directly sensable with little or no cost and which may be continuously sensed are considered *automatic fluents*, such fluents may be directly referenced in the plan (e.g. a start condition for a job can involve an automatic fluent). Reasoning about fluents which are not directly sensable and attainment of knowledge goals is achieved by using the approach of *epistemic fluents* where for world fluent values which are not directly observable a plan variable is created which represents the value of that fluent in the real world. For example if actions need to be predicated on the value of the database internal state (the world fluent *dbstate* which is not directly observable), the planner can create a plan variable called *i_dbState* which represents the value of the fluent in the real world. This variable can then be used to control action execution or used as an action parameter. These are similar to the concept of *runtime variables* used by the Puccini planner. A goal to gain knowledge of the database internal state would be represented during plan construction 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 and reasoning about using standard causal reasoning (without the need for use of modal logic as is used for example in the PKS planner).

4 ACTION LOGIC

The planning problem is the problem of defining the initial state of the plan such that the combined plan and world system evolves so that a state satisfying the goal conditions occurs on the trajectory of every possible initial contingency. In order to construct such a plan, the planner requires an action logic which is able to perform forwards temporal projection based upon the initial plan and world state, to do this every job command must have a corresponding set of event descriptions which define what the start and completion effects of the command execution is on both the plan state (job status, planner variables) and fluents in the external world. Exogenous events must also be reasoned about. The event definitions used by the action logic are STRIPS (Fikes, 1971) style descriptors with conditions which must hold for that event to occur and the effect conditions produced by that event. These *Triggered events* are defined by their trigger state descriptor and effect state descriptors. The meaning of the event descriptors is different from other planners since all the events are considered as triggered events - the occurrence of a state in which all the trigger conditions hold is not just a prerequisite for that event but it *entails* occurrence of the event and its effect conditions. See table 1 for an example set of event definitions for a job.

A state descriptor consists of a unique name (used for readability purposes during inferences) and a set of fluent conditions which hold in that state (similar to the state definition used in the fluent calculus (Thielscher, 1999). Predefined names are reserved for *currentState* and *goalState*. The conditions which hold in a state are denoted by a set of condition predicates $Hold(currentState, condition)$ where *condition* is a condition on a fluent (either internal or external) e.g. $freeDiskSpace \geq 5000$. Fluents not included in the effect state definition, retain the same value as prior to the event. In this approach, *all* changes (including those arising from agent actions) are considered as a result of triggered events. In the dynamic model all event effects are deterministic and all uncertainty is represented in the initial state descriptor (any condition not specified as holding in the *currentState* is not determined). Contingencies are defined as a sub-state of the current state defined by a further set of conditions which hold, beyond those that hold in the current state. e.g. $Contingency(dbState = 0)$.

Table 1: Start, Successful and Failure Event definitions for command *"genReport ?date"*.

Trigger conditions	Effect conditions
$status = Initialised$	$status = Executing$
$status = Executing \wedge$ $inputFile.exists = True$	$status = Completed \wedge$ $report?date.exists = True \wedge$ $report?date.contents = ?date \wedge$ $report?date.location = localServer$
$status = Executing$ $inputFile.exists = False$	$status = Completed$

Event occurrence is defined as the first occurrence of its trigger state, not by a time value (as in the event calculus (Shanahan, 2000)). The advantage of this is

that the same defined event can can occur at different times on different contingencies. This allows for reasoning about plan branch merges where an event may draw its support from different causal sources under different contingencies (which might involve the event happening at different times under those different contingencies). Inferences exist to consolidate proven occurrences on different contingencies so for example if a state occurrence is proven on the contingency where $dbState = 0$ and on the contingency where $dbState \neq 0$ then the state occurrence is proven on the trajectory of the current state.

The predicate *Occurs* (one of whose arguments is a contingency descriptor) is used to reason about occurrences of *trajectory predicates* such as conditions, states, events and partial order planning predicates such as event orderings, protections (Weld, 1994) under different contingencies. The *Occurs* predicate is true when the specified trajectory predicate occurs on all trajectories of any initial state which belongs to that contingency.

Some of the key action logic inferences are shown in tables 2, 3 and 6.

Table 2: State update inference.

$Occurs(StateOn(triggerState), contingency)$ $\Longrightarrow$ $Occurs(StateOn(effectState), contingency) \wedge$ $Occurs(OrderingOn(triggerState, effectState), contingency)$

Table 3: Causal support inference.

$Occurs(StateOn(stateA), contingency1) \wedge$ $HoldsInState(stateA, conditionA) \wedge$ $(conditionA \Longrightarrow conditionB) \wedge$ $Occurs(ProtectionOn(stateA, stateB, conditionA), contingency2) \wedge$ $Occurs(OrderingOn(stateA, stateB, contingency3)$ $\Longrightarrow$ $Occurs(StateConditionOn(stateB, conditionB), (contingency1 \cap contingency2, \cap contingency3))$

4.1 Partial Order Action Logic Inferences

Since the actions have variable length durations, a linear planning approach cannot be followed and in order to perform forwards and backwards temporal projection, the action logic must support partial order planning predicates such as causal support, orderings and protection of conditions between events. Backwards inferences must include resolution of threats using the techniques of promotion, demotion and separation (Pryor and Collins, 1996) All of these forms of reasoning must be supported in a contingent manner - hence there are trajectory predicates defined to allow contingent reasoning about occurrences of all of these.

Due to space considerations the inferences can only be sketched in this paper, but the worked examples are intended to illustrate some of the key inferences.

5 IMPLEMENTED PLANNER

The planner approach for reasoning with this action language is taken is that advocated by (Stone, 1998), (Shanahan, 2000), of planning as an abductive inference process. The agent uses backwards inferences which make abductive choices about the jobs and plan variables it places into the plan. Once a choice about the plan components has been made, the planner performs forwards inferences to determine the evolution of the plan over time under different contingencies and the plan is considered as complete once it has been proved that the goal state occurs on all possible contingencies. The planner was implemented using the drools rules engine (JBoss, 2007), each inference in the action logic corresponding to a production rule. Drools contains an automated logical retraction facility which was used to implement search backtracking.

6 WORKED EXAMPLES

6.1 Handling Exogenous Events, Action Monitoring and Triggered Actions

In this example (solved by the implemented planner) from the previous discussion , the goal is to produce a report file *"remoteReport*1220" 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 event definitions for these are shown in tables 1, 4 and 5. Note, 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

Table 4: Start, Success, and Failure event definitions for action "$ftpToRemote\ ?file$".

Trigger conditions	Effect conditions
$status = Initialised$	$status = Executing$
$status = Executing \wedge$ $?file.exists = True \wedge$ $?file.location = localServer \wedge$ $?file.contents = ?contents$	$status = Completed \wedge$ $remote?file.exists = True \wedge$ $remote?file.location = remoteServer \wedge$ $remote?file.contents = ?contents$
$status = Executing$ $?file.exists = False$	$status = Completed$

Table 5: Event definition for exogenous event *externalFileGen*.

Trigger conditions	Effect conditions
$inputFile.exists = False$	$inputFile.exists = True$

$remoteReport1220.exists = True$ by adding into the plan a new job to run the action $ftpToRemote$ with the parameter $?file =$ "$Report1220$" and instantiating all its associated events (start event, success event, fail event). It then sets subgoals to prove that the ftp start event is triggered, that the $remoteReport1220.exists = True$ is protected from the successful ftp event effect state to the goalState and that successful event effect is ordered before the goal state. Similar subgoals are created for the location conditions.

Because a file existence is considered as an automatically sensed fluent, the planner inserts a start condition for the $ftpToRemote$ job that $Report1220.exists = True$ in order for the job to start - this becomes part of the trigger state definition for the ftp start event - which means that the ftp command will not be executed until the condition $Report1220.exists = True$ is true.

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

Using the same forms of inference to provide support for the condition $Report1220.exists = True$ the planner creates a new job 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 $genReportJob.status = Complete$ to the start conditions for $ftpToRemote$.

Support for the $inputFile.exists = True$ trigger condition for *genReportJob* is obtained from the event *externalFileGen*. Since the *externalFileGen* trigger state has no conditions, using the inference "All state conditions proven then state occurrence proven" the planner is able to prove occurrence of the *externalFileGen* event on all trajectories.

Table 6: All state conditions proven then state occurrence proven.

$$\forall condition \in Condition,\ contingency \in Contingency\ s.t. HoldsInState(state, condition) \wedge Proven(Occurs(StateConditionOn(condition, state), contingency)) \Longrightarrow Proven(Occurs(StateOn(state), contingency))$$

From the occurrence of *externalFileGen*, using a series of forwards inferences, including the "State update inference", "All state conditions proven then state occurrence proven", and other partial order inferences using orderings and protections the planner is able to prove occurrence of the events *externalFileGen*, *genReportJob* success, $ftpToRemote$ success and occurrence of the *goalState* on all trajectories.

The complete plan consists of the following:

```
(name: genReport1220,
command:"runReport 1220" ,
status:Initialised,
startConditions: inputFile.exists=True)
(name:ftpToRemote_Report1220,
command: "ftp Report1220",
status:Initialised,
startConditions: Report1220.exists=True,
   genReport1220.status = Complete)
```

6.2 Planning with Knowledge Goals and Merged Contingencies

This example (under which the planner implementation is currently being evaluated) is from the repair database error scenario previously discussed 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* ?*e*" which checks whether the database has a particular error *e* (a value of 1 or 2) The knowledge acquisition part of this problem is analogous to the standard knowledge planning problem *Safe combination problem* (Petrick and Bacchus, 2002)).

The event schema definition for the internal plan variable assignment action *assign*, the *checkDB* com-

mand and *repairDB* action are shown in tables 7,8 and 9.

Table 7: Event definition for command "*assign* ?*x* ?*y*".

Trigger conditions	Effect conditions
$status = Initialised$	$status = Completed \wedge (?x = ?y)$

Table 8: Event definition for command ?*result* = "*checkDB* ?*e*".

Trigger conditions	Effect conditions
$status = Initialised$	$status = Executing$
$status = Executing$	$status = Completed \wedge (?result = (dbState = ?error)$

Table 9: Start, Success and Failure Event definition for command "*repairDB* ?*error*".

Trigger conditions	Effect conditions
$status = Initialised$	$status = Executing$
$status = Executing$ $dbState = ?e$	$status = Completed \wedge dbState = 0$
$status = Executing$ $dbState \neq ?e$	$status = Completed$

In the initial state : $Holds(currentState, dbState = 1|2)$ (where the "|" signifies that the value held is one of the specified values)

The goal is to repair the database error : $Holds(goalState, dbState = 0)$

In order to provide support for this condition the planner creates a repair job and an epistemic program variable *i_dbState* which is specified as the parameter for the repair action. It then creates a subgoal that $i\_dbState = dbState$ at the time the repair job is run (since the repair action only works if its runtime parameter is equal to the database error)

The planner establishes support for the condition $i\_dbState = dbState$ by adding a job with the assignment command "*assign i_x i_y*" with the substitutions $i\_x = i\_dbState$, $i\_y = 1$ and with the additional *required assumption* that the initial contingency is one where the condition $dbState = 1$ holds. This proves causal support for $i\_dbState = dbState$ but only on the contingency $dbState = 1$. It similarly proves causal support from "*assign i_dbState* 2" on the contingency $dbState = 2$.

The planner must prevent the threats that these jobs pose to each other's support of $i\_dbState = dbState$. It performs this threat resolution via *separation* (Pryor and Collins, 1996) whereby a threat by a conflicting action is resolved by ensuring that the threatening action does not occur in the same contingency where the threatened causal support is needed. The planner determines that the "*assign i_dbState* 1" command must occur on the contingency where $dbState = 1$ and it must not occur on the contingency where $dbState \neq 1$

In order to provide the necessary conditioning for the "*assign i_dbState* 1" job the planner introduces contingency control on this action by creating a new boolean plan variable *i_dbStatus_is_*1 which represents the value of the truth/falsity of the proposition $dbState = 1$. It adds truth of this planner variable as a start condition to the "*assign i_dbState* 1" job.

The planner establishes the value of *i_dbStatus_is_*1 using the sensing action command "$i\_dbState\_is\_1 = checkDB$ 1"

To prove ordering of the "*repairDB i_dbState*" command to after the epistemic fluent has been correctly set the planner adds $i\_dbState \neq null$ to the start conditions to ensure it occurs after *i_dbState* has been set.

From forwards inferences the occurrence of $i\_dbState = dbState$ in the trigger state for "*repairDB i_dbState*" success event is proven on the contingency $dbState = 1$ and similarly it is also proven to occur on contingency $dbState = 2$. Using an inference which combines proven occurrences of the same event across different contingencies, the planner proves the occurrence $i\_dbState = dbState$ in the *repairDB i_dbState* trigger state on all the trajectories of the current state. From this it is able to prove occurrence of the successful repair action and the goal state on the trajectory of the current state.

The final plan is:

```
(name: i_dbState, value: null )
(name: i_dbState_is1, value:null )
(name: i_dbState_is2, value:null)

(name:check1,
command: i_dbState_is1="checkDB 1" ,
status:Initialised, startConditions:)

(name:check2,
command: i_dbState_is2="checkDB 2" ,
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:repairDB,
command: "repairDB i_dbState" ,
status:Initialised,
startConditions: i_dbState not null )
```

7 CONCLUSIONS AND FUTURE WORK

In this paper we have presented a new contingent plan representation which offers advantages with respect to action monitoring, handling of triggered events, compactness of plan branch representation and which can handle planning for knowledge goals. We have briefly sketched how a planner is able to reason about this plan representation and how it can generate plans for some key domain scenarios. An implementation has been successfully demonstrated on the first example, and is currently being evaluated against the second example. A further evaluation is planned on an example where the value of two independent fluents must be sensed - demonstrating that search time scales linearly with the number of sensed fluents - (not exponentially as is the case with a planner which does not allow execution branch remerging.) Future extensions to the planner could be made by the introduction of other inference rules - for example temporal inferences by reasoning about time conditions - and by the introduction of specialised inferences to build more complex predesigned plan structures - with validation of the constructed plan using the forwards dynamical inferences.

Although created for the computer batch job domain the representation could be applied in any domain where actions are triggered in response to external events, this includes workflows and event driven architectures. It is hoped that this representation will undergo further future development and application to such domains.

REFERENCES

Bertoli, P., Cimatti, A., Pistore, M., Roveri, M., and Traverso, P. (2001). Mbp: a model based planner. In *Proceedings of the IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information, Seattle*.

Bonet, B. and Geffner, H. (2001). Gpt: A tool for planning with uncertainty and partial information. In *In Proc. IJCAI01 Workshop on Planning with Uncertainty and Incomplete Information*, pages 82–87.

ComputerAssociates (2002). Autosys.

Draper, D., Hanks, S., and Weld, D. (1994). Probabilistic planning with information gathering and contingent execution. pages 31–36. AAAI Press.

Ennis, R. (1986). A continuous real-time expert system for computer operation. *IBM J. research development*, 30(0):0.

Fikes, N. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artifical Intelligence*, 2:189–208.

Golden, K. (1998). Leap before you look: Information gathering in the puccini planner. In *Proceedings of AIPS*, pages 70–77.

Grosskreutz, H. and Lakemeyer, G. (2000). cc-golog: Towards more realistic logic-based robot controllers. In *In AAAI'2000*, pages 476–482.

JBoss (2007). Drools 4.0.7.

Levesque, H. J., Reiter, R., Lesperance, Y., Lin, F., and Scherl, R. B. (1997). GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83.

Murch, M. (2004). *Autonomic computing*, chapter Introduction. IBM Press.

Petrick, R. and Bacchus, F. (2002). A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of AIPS'02*, pages 212–221.

Pryor, L. and Collins, G. (1996). Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, 4:287–339.

Shanahan, M. (2000). An abductive event calculus planner. *Journal of Logic Programming*, 44:207–239.

Stone, M. (1998). Abductive planning with sensing. *AAAI*.

Thielscher, M. (1999). From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence*, 111:277–299.

UC4 (2008). Application automation in enterprise workload automation.

Weld, D. S. (1994). An introduction to least commitment planning. *AI Magazine*, 15(4):27–61.