Planning in Multi-Agent Systems

Salvador Jacobi

Kongens Lyngby 2014
Compute-BSc-2014

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk Compute-BSc-2014-1

Summary (English)

The goal of the thesis is to investigate how the GOAL agent programming language can be extended with automated planning capabilities. Agent programming was partially motivated by the lack of flexible planners, but agent programming languages often lack useful planning capabilities. A prototype of a planning module that uses the partial-order planning algorithm is implemented in Java and integrated into the GOAL codebase.

Summary (Danish)

Målet for denne afhandling er at undersøge hvordan agent-programmeringssproget GOAL kan udvides med muligheder for automatiseret planlægning. Agent programmering var partielt motiveret af mangel på fleksible planlægningværktøjer, men agent-programmeringssprog mangler ofte nyttige planlægningsmuligheder. En prototype af et planlægningsmodul som benytter partial-order planlægnings algoritmen er implementeret i Java og integreret i GOAL's codebase.

Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfilment of the requirements for acquiring an BSc in Software Technology.

Lyngby, 01-July-2014

Salvador Jacobi

Contents

Chapter 1

Introduction

The purpose of this project is to integrate automated planning into the GOAL agent programming language. This will enable GOAL agents to compute a plan—a sequence of actions—that will achieve a particular goal without the need to define a rule based strategy in the agent program that is specific to the problem domain.

My task has been to implement a planning module that extends the GOAL programming language. The GOAL codebase is written Java and my planning module is implemented by extending this codebase to allow users of the GOAL programming language to invoke this planning module.

My thesis is organized as follows. I will start with an introduction of the GOAL programming language and some of the relevant concepts in automated planning. I will then discuss the planning module and how it integrates into GOAL. I will also describe the partial-order planning algorithm and my attempt to implement it in Java. Finally, I will conclude this project.

Chapter 2

The GOAL Agent Programming Language

GOAL is a programming language and a platform designed specifically for programming cognitive agents and simulating their behavior in a multi-agent system. The term *agent* is used to describe an entity that can perceive information about its environment and perform actions that can potentially change the state of this environment. GOAL provides tools to program the behavior of agents that can act and perceive in an environment.

An example of an environment could be a version of the famous blocks world domain. In the blocks world domain blocks are arranged on a table and the blocks can be sitting either directly on the table or on top of another block. There is always room on the table for more blocks, but there can only be one block directly on top of another. A block can be moved onto the table or onto another block, but only one can be moved at a time. This could be extended with multiple agents that are all capable of moving blocks around. Moving a block from one position to another would be an action that is then broadcasted to the other agents as a percept about who moved what block to and from where. Agents could work together or against each other to achieve some desired configuration of the blocks.

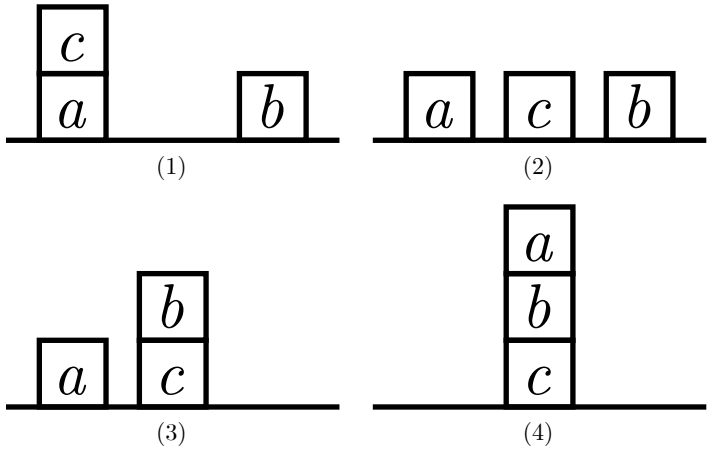


Figure 2.1: Blocks world configurations

> Here's an example of the blocks world domain with three blocks a, b, and c that are moved from one configuration to another over the course of three move actions.

A GOAL agent maintains a *mental state*, which is all the information the agent currently has about the environment and the goals it wants to achieve. This information is represented using a *knowledge representation language* (KR-language), which is a symbolic, logic language. GOAL is not retricted to any specific KR-language, but the important thing is that the represented information can be updated and queried through the interface that GOAL defines. The currently implemented KR-language is Prolog, which will be used in the following examples.

The mental state of an agent is separated into three categories; knowledge, beliefs, and goals. The knowledge base is static and cannot be changed during execution. It holds general knowledge about the environment, such as rules and facts that always apply. The belief base is dynamic and can be updated during execution to reflect the current state of the environment. This is everything specific to the current state of the environment. The goal base contains goals that the agent wants to achieve. These should ideally be declarative goals that specify what the state of environment should be, but nothing about how to achieve it. When a goal is achieved (i.e. when its thruth value can be derived from the knowledge and belief bases) it is removed from the goal base. The goal base is also dynamic and the agents can *adopt* and *drop* goals during execution.

An environment is an external program that communicates with the GOAL runtime through the Java-based Environment Interface Standard (EIS). Agents interact with the environment through *percepts* and *actions*.

A percept is sent by the environment to a specific agent and contains information about some aspect of the environment that the agent can perceive. This information could be anything about the state of some simulation or even data from a hardware sensor. Agents process incoming percepts to update their beliefs about the environment by inserting and/or deleting facts into/from the belief base.

Agents can perform actions to potentially change the state of the environment. Available actions are formally specified in the agent program. An action specification consists of the action name and its parameter variables, a *precondition*, and a *postcondition*. A precondition specifies under what conditions an action can be performed and a postcondition specifies the effects of performing an action. When the precondition for an action is satisfied, the action is said to be *enabled*. When an action is performed, the belief base is updated according to the postcondition.

```
move(X,Z) {
  pre { block(X), on(X,Y), clear(X), clear(Z) }
  post { on(X,Z), not( on(X,Y) ) }
}
```

Figure 2.2: Action specification of move action

> Here is an example of an action specification in the blocks world domain that moves a block X onto Z (either a block or the table). The precondition states that X should be a block, that X should be on top of something Y, and that both the block being moved and the destination is *clear*, i.e. there's nothing on top of it (except for the table which is always clear).

A GOAL agent program is separated into modules that consist of different sections. There are three special modules: `init`, `main`, and `event`. The init module is used to specify the initial knowledge, beliefs and goals of the agent as well as for defining the action specifiations. The main module serves as the entry point of the agent program, and the event module is used for processing percepts. It is also possible to specify custom modules that can be invoked with an *action rule*.

```
knowledge {
  block(a).
  block(b).
  block(c).
  clear(table).
  clear(X) :-
    block(X),
    not( on(_,X) ).
}
```

```
beliefs {
  on(c,a).
  on(a,table).
  on(b,table).
}
```

```
goals {
  on(a,b), on(b,c).
}
```

Figure 2.3: Knowledge, beliefs, and goals

This is an example of the initial knowledge, beliefs, and goals of an agent in the blocks world domain. The knowledge section encodes the following information. There are three blocks named a, b, and c. The table is always *clear*, i.e. it's always possible to put something on the table. The rule *clear(X)* says that a block is clear if there is nothing on top of it. The beliefs section encodes the following initial configuration of blocks. The blocks are configured as shown in fig. 2.1.(1). The goals section specifies a single goal—that the blocks should be stacked as shown in fig. 2.1.(4).

The main module must contain a `program` section. A program section defines a strategy for selecting actions to perform using *action rules*. A program section consists of a list of action rules of the form **if** ψ **then** α, where ψ is a *mental state condition* that specifies when the action α (or combination of actions) is *applicable*. A mental state condition is a query that inspects the knowledge, belief, and goal bases of the agent. It consists of *mental atoms*, such as **bel(**ϕ**)** and **goal(**ϕ**)**, that can be combined in a logic expression.

```
if goal( clear(Y) ), bel( on(X,Y) ) then move(X,table).
```

Figure 2.4: Action rule

Here's an example of an action rule where the mental state condition is a conjunction of two mental atoms. The **goal** atom inspects the goal base to see if we have a goal to clear Y. The **bel** atom inspects the combination of the knowledge and belief base to see if we currently believe that some block X is on top of Y. The action moves the block X away from Y, however only if there's nothing on top of X.

When the action of an *action rule* is both *enabled* and *applicable* it is called an *option*. An action rule will generate an option for every configuration of action parameters that satisfy the mental state condition.

The order in which action rules are considered during execution is determined by the *rule evaluation order* of the containing program section. The possible evaluation orders are `linear`, `random`, `linearall` and `randomall`. With the `linear` order, action rules are evaluated from top to bottom and the first option generated will be performed. With the `random` order, an option is randomly selected from all the options generated by all the rules. The `linear` and `random` orders will only perform one action (or action combo) in each evaluation step of a module, whereas the `linearall` and `randomall` orders will consider all the rules and perform multiple actions in the same evaluation step. There's also an `adaptive` order that invokes an adaptive learning algorithm that is then responsible for selecting an option.

A module will continue evaluating until its *exit condition* holds. There are four possible exit conditions:

`nogoals` — exit if there are no goals left in the goal base

`noaction` — exit if no action was performed in this evaluation step

`always` — always exit the module

`never` — never exit the module

Note that there is a special action `exit-module` that will exit a module regardless of its exit condition. The default exit condition is `always`, except for the main module, where it's `never`.

When a GOAL program is started, the main module is evaluated. Whenever an action is performed a *cycle* is started that fetches new percepts from the environment, evaluates the init module if it hasn't been already, and then evaluates the event module. A cycle is also started if no action was performed after evaluating the main module. The agent program terminates when the main module exits.

Chapter 3

Automated Planning

"Planning is the reasoning side of acting. It is an abstract, explicitly deliberation process that chooses and organizes actions by anticipating their expected outcomes." [GNT04]

Planning can be viewed as a search in a transition system $\Sigma = (S, A, \gamma)$, where S is a finite set of all possible states of the environment (or of a simplified model thereof), A is a finite set of all possible actions, and $\gamma : S \times A \to S$ is a state-transition function that will compute the resulting state of performing an action. This is assuming that the transition system is *deterministic*, i.e. the result of performing an action in some particular state will always result in the same state. A transition system Σ can be represented as a directed graph where the vertices are the states in S and the arcs are actions in A, such that the action a will go from state s to s' if and only if $\gamma(s, a) = s'$.

The planning problem can then by defined as a triple $\mathcal{P} = (\Sigma, s_0, g)$, where Σ is a transition-system, s_0 is the initial state, and g is a set of goal states. A solution to the planning problem would then be a sequence of actions that form a path in the graph from s_0 to $s \in g$.

One of the main issues is how to define $\mathcal{P}$ without explicitly enumerating all the possible states and actions, as the number of these quickly become very large due to combinatorial explosion. For example, there are $3.27697927886085654441 \cdot$

10^{20} distinct states in blocks world domain with just 20 blocks[1].

One method of implicitly representing $\mathcal{P}$ is the classical representation, which uses a first-order language L with a finite number of predicate and constant symbols and no function symbols. A state is then defined as a set of ground atoms of L. These are all the predicates that hold in that state. Predicates not mentioned are considered false (this is the closed world assumption (CWA)).

Actions are represented using operators. An operator is a template for a set of actions. It consists of an action name that includes parameter variables, a *precondition* and the *effects* that are both sets of literals (an atom of the negation thereof) that may contain the variables mentioned in the parameters. An action is then a ground instance (no variables) of an operator. The state transition function then be defined as follows: $\gamma(s,a) = (s \backslash \text{effects}^-(a)) \cup \text{effects}^+(a)$, where $\text{effects}^+(a)$ and $\text{effects}^-(a)$ are the atoms of resp. the positive and negative literals that appear in the effects of an action a.

The *statement* of a planning problem is what corresponds to syntical specification, e.g. how it would be represented in the GOAL language. The statement of the planning problem $\mathcal{P} = (\Sigma, s_0, g)$ is defined as $P = (O, s_0, g)$, where O is the set of operators.

[1] This happens to follow the integer sequence `http://oeis.org/A000262`, which describes the number of "sets of lists".

Chapter 4

Planning Module

The planning module is an extension to the GOAL programming language. When the planning module is invoked by an agent, it selects a single goal from the goal base and attempts to find a plan to achieve this goal. If successful, the planning module will have the agent perform the actions of the plan, one at a time.

One way of achieving this would be to integrate an external PDDL planner by translating the planning problem in GOAL into PDDL and have the external planner find a plan that is then brought back into GOAL for execution. PDDL stands for Planning Domain Definition Language and is an attempt at standardizing the description of planning domains and problems.

Another solution would be to write a planner myself that is part of the GOAL codebase. This would also allow me to make use of data structures and utilities already defined in GOAL when implementing the planning algorithm.

I decided to write a partial-order planner based on the PoP procedure defined in [GNT]. One reason for choosing a plan-space planning algorithm (as opposed to state-space) is that it lends itself well to multi-agent planning, as the partial plan structure allows for plan-merging operations to be defined and handled more easily. The planning problem could be decomposed and distributed to agents and then finally merged. I did not have the time to look into this aspect

multi-agent planning, but making the choice for a plan-space planner would make it easier to later implement such a feature in GOAL.

I have tried my best to keep the planner independent of the KR-language by only using the interfaces defined in GOAL. There are, however, subtle aspects such as variable renaming that might depend on the KR-language in use, e.g. in Prolog variables must be capitalized.

4.1 Simplifying assumptions

Predicates that derive from other predicates introduce a lot of difficulties when planning, so I have decided to only consider basic predicates. One way of dealing with rules would be to compile them into simple facts before invoking the planner. I decided to simply assume that the agent program makes no use of rules, such as the *clear(X)* predicate mentioned earlier.

Another assumption I've made is that all action specifications should adhere to the STRIPS format, where preconditions are conjunctions of *positive* literals and postconditions are conjunctions of literals. This limits the expressiveness of the precondition as it is no longer possible to specify that certain predicates should *not* hold for the action to be enabled. I also assume that all variables mentioned in the precondition and postcondition appear in the parameters of the action specification. This is not something GOAL requires, but will make it easier to deal with variables in action specifications.

4.2 Integration

The planner is integrated into GOAL by implementing a new rule evaluation order. In a similiar way to `adaptive` order that invokes an adaptive learning algorithm, I have implemented a `plan` order that invokes my planning algorithm that then performs the actions of the resulting plan (if any). The planning module hooks into the GOAL runtime inside the `run` method of the `RuleSet` class with an addition case to a switch statement (see line 21 in the appendix).

The program section of a module with the `plan` order is completely ignored, as everything is left to the planner. The action rules of the program section could be used to aid the planner with information specific to the domain, but this is not something I've looked into.

The planner works by extracting the facts of the knowledge and belief bases and combining them into a set of `DatabaseFormula`, which is a Java interface implemented by the KR-language plugin (Prolog in this case) that represents an expression that can be inserted into a database of expressions. This corresponds to the initial state s_0 of the planning problem. The goal is also extracted as a set of `DatabaseFormula` which then corresponds to g. The action specifications of the init module are gathered as soon as the agent program has been parsed, and the preconditions and postconditions of actions are also converted to sets of `DatabaseFormula`.

If the planner finds a plan, it will perform the actions of the plan on at a time until all action have been performed or the precondition of the next action doesn't hold. If the planner returns failure, then the module will simply do nothing.

If the planner is unable to find a solution plan it might keep on searching for a very long time, stalling the agent program. One way to deal with this would be to implement some kind of time out period that simply stops the planner in case it takes too long as well as some precaution that prevents it from happening again immediately. This is not something I have implemented and there are many other refinements that can be made. It would be useful if the programmer could also specify some kind of strategy for when to give up planning or when the replan in case of an unsatisfied precondition caused by other agents changing the state of the environment while the planner is running.

4.3 Test program

In order to test the correctness of my planning module I set up a simple test program that solve a problem based on the blocks world domain (See appendix for a listing of the `plan.goal` program) The example given earlier used a rule to determine if a block was clear. In order to work with the planning module the rule must be eliminated. The clear rule is removed from the knowledge base and instead `clear(c)` and `clear(b)` is added to the belief base.

```
knowledge {
  block(a).
  block(b).
  block(c).
  clear(table).
}
```

```
beliefs {
  on(c,a).
  on(a,table).
  on(b,table).
  clear(c).
  clear(b).
}
```

```
goals {
  on(a,b), on(b,c).
}
```

Figure 4.1: Knowledge, beliefs, and goals

In order to update the clear predicate when the blocks are moved, the move action is split into two actions; `move1(X,Y,Z)` for moving a block X sitting on Y onto a another block Z and `move2(X,Y)` for moving a block X sitting on Y on to the table. The postconditions will update the clear predicate to correctly reflect the environment.

```
move1(X,Y,Z) {
  pre { block(X), block(Z), on(X,Y), clear(X), clear(Z) }
  post { on(X,Z), not( on(X,Y) ), clear(Y), not( clear(Z) ) }
}

move2(X,Y) {
  pre { block(X), on(X,Y), clear(X) }
  post { on(X,table), not( on(X,Y) ), clear(Y) }
}
```

Figure 4.2

It's trivial to see that the *optimal* plan (the plan with the least actions) for stacking the blocks a, b, and c on top of eachother is the following sequence of actions: ⟨move2(c, a), move1(b,table,c), move1(a,table,b)⟩.

The main module simply contains a program section with the rule evaluation order `plan`. The program section contains a single dummy action rule that is never actually evaluated with the only purpose of keeping the GOAL parser happy.

Chapter 5

Partial-Order Planning Algorithm

The partial-order planning (POP) algorithm is a plan-space planning algorithm as opposed to a state-space planning algorithm. In state-space planning you search the the graph of Σ to find a goal state. In plan-space planning you search in a graph where the vertices are (potentially unfinished) plans and the arcs are *refinement operators* that step by step fill out the details of a plan.

The POP algorithm works on *partial-order* plans. This means that the actions of a plan do not have to be in one specific sequence (i.e. in *total-order*). Instead, actions and orderings are decoupled and *ordering constraints* are placed on pairs of actions. An ordering constraint simply tells wether some action should come after another action, but not necessarily *directly* after it. A partial-order plan can thereby represent multiple total-order plans, i.e. all the possible interleavings of actions that satisfy the ordering constraints. The algorithm follows the *least commitment strategy*, where only the strictly necessary constraints are put on a plan in order to progress, e.g. actions of the plan are partially instatiated such that variables can remain unbound and action are only ordered when necessary.

Consider the statement $P = (O, s_0, g)$ of a classical planning problem $\mathcal{P}$. A partial-order plan is a tuple $\pi = (A, \prec, B, L)$, where A is a set of partially

instantiated operators of O (referred to as actions), $\prec$ is a set of ordering constraints $a_i \prec a_j$ that tells that action a_i is ordered before a_j, B a set of binding constraints on the variables of the actions of A, and L is a set of causal links $\langle a_i \xrightarrow{p} a_j \rangle$ that tells that action a_i provides the precondition p for the action a_j.

A partial-order plan is a solution to the planning problem if it has no *flaws*. A *flaw* is either a *subgoal* or a *threat*, and these are related to the causal links of L. A causal link $\langle a_i \xrightarrow{p} a_j \rangle$ specifies that the action a_i has effect p that satisfies the precondition p of the action a_j. A *subgoal* is a precondition without a corresponding causal link. If an action a with an effect $\neg q$ can be interleaved in-between two actions a_i and a_j of a casual link $\langle a_i \xrightarrow{p} a_j \rangle$ and p can be unified[1] with q, then a is said to be a *threat* on the causal link, as it could potentially negate an effect that one action provides for another.

The POP algorithm is started with the *empty* plan. This plan contains two *pseudo* actions a_0 and a_∞ that do not map to any actions that can be performed by an agent. The *start* action a_0 has an empty precondition and effects s_0. The *finish* action a_∞ has no effects and precondition g. The plan also contains the ordering $a_0 \prec a_\infty$ and all actions inserted later will be ordered in-between a_0 and a_∞ such that a_0 and a_∞ are resp. the first and last action. This plan obviously contains flaws, as the action a_∞ contains subgoals.

The idea is then to keep refining the plan until there are no flaws are left. This is acheived by adding causal links to solve subgoals as well as new actions to create these casual links. Threats are resolved by adding either ordering or binding constraints such that the threat is avoided. The threatening action a can either be *demoted* or *promoted* with resp. $a_j \prec a$ and $a \prec a_i$. Another way to resolve the threat is to add a binding constraint such that p and q are no longer unifiable.

The PoP algorithm is shown in alg. 1. It takes two arguments π and agenda. The first argument π is the current version of the plan. The agenda is a set of ordered pairs (a, p), where a is an action of A and p is a precondition of a that is also a subgoal. When the agenda is empty, there there's nothing left to do and the solution plan is returned.

The algorithm is initially called with the empty plan and an agenda that contains (a_∞, p) for all preconditions p of a_∞.

[1] Two terms can be unified if there exists a substitution such that the terms become equal, e.g. the substitution $\{X \mapsto \mathrm{a}, Y \mapsto \mathrm{table}\}$ unifies the two terms $\mathrm{on}(X, \mathrm{table})$ and $\mathrm{on}(\mathrm{a}, Y)$.

Algorithm 1 PoP algorithm as given in [GNT04]

```
1: procedure POP(π, agenda)                          ▷ where π = (A, ≺, B, L)
2:     if agenda = ∅ then
3:         return π
4:     end if
5:     select any pair (a_j, p) in and remove it from agenda
6:     relevant ← PROVIDERS(p, π)
7:     if relevant = ∅ then
8:         fail
9:     end if
10:    non-deterministic choice of action a_i ∈ relevant
11:    L ← L ∪ {⟨a_i --p--> a_j⟩}
12:    update B with the binding constraints of this causal link
13:    if a_i is a new action in A then
14:        update A with a_i
15:        update ≺ with (a_i ≺ a_j), (a_0 ≺ a_i ≺ a_∞)
16:        update agenda with all preconditions of a_i
17:    end if
18:    for each threat on ⟨a_i --p--> a_j⟩ or due to a_i do
19:        resolvers ← set of resolvers for this threat
20:        if resolvers = ∅ then
21:            fail
22:        end if
23:        non-deterministic choice of resolver in resolvers
24:        add that resolver to ≺ or to B
25:    end for
26:    return POP(π, agenda)
27: end procedure
```

The PROVIDERS(p, π) procedure returns actions that are relevant to the precondition p. These are new actions or actions already in A that have an effect that could potentially satisfy the precondition p.

There are two non-deterministic choices for selecting actions and resolvers. Backtracking over these non-deterministic choices works the following way. When a **fail** is encountered, the algorithm continues from where the previous non-determinstic choice was made with a new choice. If all choices are exhausted it counts as a **fail** and it backtracks even further. If there is no previous non-determinstic choice the algorithm fails and no plan is found.

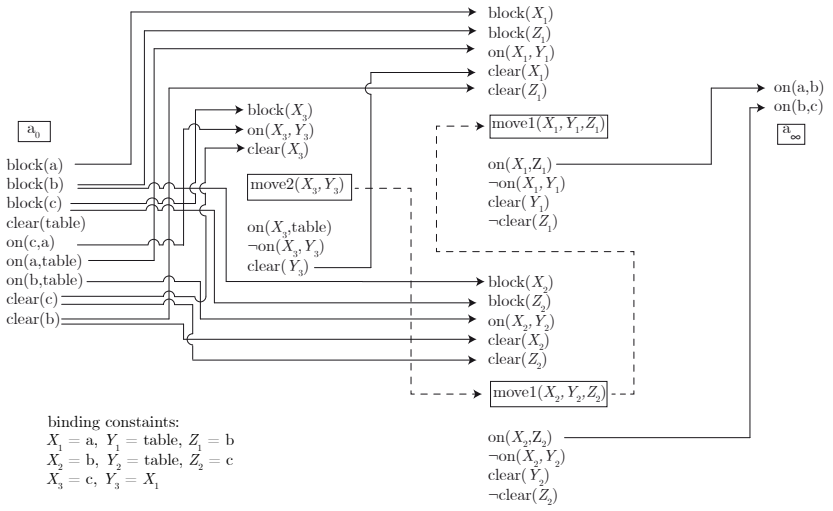


Figure 5.1: Graphical representation of a partial-order plan

Fig 5.1 is a graphical representation of a partial-order plan that solves the blocks world problem shown in fig. 4.1. The boxes are the actions of A with precondition above and effects below. Casual links are shown with solid arrows and some ordering constraints are explicitly shown with dashed arrows. The majority of the ordering constraints are implicit: $a_0 \prec a$ for all $a \neq a_0$, $a \prec a_\infty$ for all $a \neq a_\infty$, and casual links imply an ordering constraint in the same direction.

This partial-order plan has only one *linearization*, i.e. it only represents one total-order plan: ⟨move2(c, a), move1(b,table,c), move1(a,table,b)⟩. The two ordering contraints shown explicitly are the result of two threat resolvers. The action move1(X_1, Y_1, Z_1) initially threatened the casual link that provides the precondtion clear(X_2) with the effect ¬clear(Z_1), as $X_2 = Z_1 = b$. This is resolved by demoting the threat action such that it is ordered after the casual link: move1$(X_1, Y_1, Z_1) \prec$ move1(X_2, Y_2, Z_2).

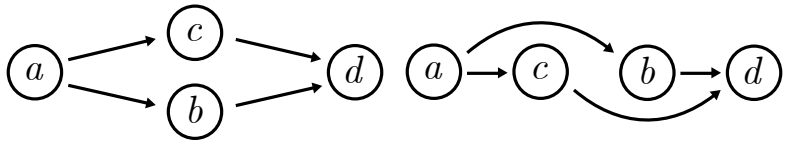
5.1 Implementation

In order to implement the PoP algorithm in Java, I have translated the pseudocode to a version that backtracks using recursion (see alg. 2). When a failure condition is encountered, the function returns a *null*-value. The algorithm works on a single plan that is updated and reverted. After the plan is updated to reflect the choice of an action or resolver, a recursive call is made. If the recursive call returns *null*, then the changes made in this recursion are reverted and a new choice is made. If the recursive call returns a non-*null* value, then it is propagated and the algorithm returns the solution plan.

The threat resolution is moved into a separate procedure POP2, that recurses to resolve threats. If all threats are resolved, it calls the main procedure POP to continue.

The ordering constraints are handled by an `OrderingManager` that maintains the transitive closure of $\prec$. This makes it possible to query ordering constraints in constant time, however, it is significantly more costly to update the orderings as it requires a re-propagation of the transitive closure with complexity $O(n^2)$. It should however be beneficial as there are usually issued more queries than updates when planning.

The `OrderingManager` class implements a method `getLinearization` that computes one of the possible total-order plans. The ordering constraints can be viewed as a *directed acyclic graph* with actions as vertices and arcs as orderings. A linearization is the found by performing a *topological sort* that puts all the actions into a specific ordering.



(a) Ordering constraints **(b)** Linearization $\langle a, c, b, d \rangle$

Figure 5.2: Example of topological sort

Algorithm 2 PoP procedure without choose/fail

```
1: procedure POP(π, agenda)                          ▷ where π = (A, ≺, B, L)
2:     if agenda = ∅ then
3:         return π
4:     end if
5:     select any pair (a_j, p) in and remove it from agenda
6:     relevant ← PROVIDERS(p, π)
7:     for a_i ∈ relevant do
8:         L ← L ∪ {⟨a_i --p--> a_j⟩}
9:         update B with the binding constraints of this causal link
10:        isNewAction ← a_i ∉ A
11:        if isNewAction then
12:            update A with a_i
13:            update ≺ with (a_i ≺ a_j), (a_0 ≺ a_i ≺ a_∞)
14:            update agenda with all preconditions of a_i
15:        end if
16:        threats ← THREATS(a_i, p, a_j)
17:        result ← POP2(π, agenda, threats)
18:        if result ≠ null then
19:            return result
20:        end if
21:        if isNewAction then
22:            remove preconditions of a_i from agenda
23:            remove (a_i ≺ a_j), (a_0 ≺ a_i ≺ a_∞) from ≺
24:            remove a_i from A
25:        end if
26:        remove binding constraints of this causal link from B
27:        L ← L \ {⟨a_i --p--> a_j⟩}
28:    end for
29:    return null
30: end procedure
31: procedure POP2(π, agenda, threats)               ▷ where π = (A, ≺, B, L)
32:    if threats = ∅ then
33:        return POP(π, agenda, threats)
34:    end if
35:    select any threat t from threats
36:    resolver ← RESOLVERS(T)
37:    for resolver ∈ resolvers do
38:        update ≺ or B according to resolver
39:        result ← POP2(π, agenda, threats)
40:        if result ≠ null then
41:            return result
42:        end if
43:        revert ≺ or B according to resolver
44:    end for
45: end procedure
```

This version of the algorithm is not very practical as the choice of the action and resolver on resp. line 7 and 37 is arbitrary. The partial-order planning algorithm is very dependent on a heuristic for the choice of actions and resolvers as this has a huge impact on plan space that must be searched. By implementing a best-first strategy where the action and resolvers are chosen based solely on their heuristic value. *Fewest alternatives first* (FAF) is the heuristic to solve the flaw (or subgoal) that results in the fewest number of resolvers. The idea is to get the flaws with the smallest branching factor out of the way as early as possible to limit the cost of eventual backtracking.

My planner implementation managed to find a plan to achieve the goal in my test program, however, this is only a coincidence. By simply changing the goal such that the blocks should be a stack in the reverse order causes the algorithm to never terminate. This is because the planner makes arbitrary choices of actions and resolvers that are not guided toward a solution. My implementation in Java can be seen in `POPPlanner.java` from line 108 in the appendix.

Chapter 6

Conclusion

I have made myself familiar with the GOAL agent programming language and the structure of its codebase in order to extend the language with planning capabilities. I have integrated a planning module that hooks in to the GOAL runtime and implemented a version of the partial-order planning algorithm in Java.

The planning module is unfortunately not in a state that makes it practical to use in a GOAL agent program, as the planning algorithm is not guided by a heuristic. It does, however, find a solution plan for the small example problem I set up and the methods for manipulating a partial plan structure appear to be working as intended.

If I had more time to work on this project, I would definitely prioritize a better search strategy that takes a heurstic function, such as FAF, into consideration. The following aspects would also be interesting to investigate:

- How to distribute a planning problem in a multi-agent system.
- What kinds of problems can be solved with a combination of planning and a rule based strategy.
- Implementing the Planning Domain Definition Language (PDDL) as a KR-language.

Appendix A

Appendix

A.1 Source code listing

src/test/resources/goal/tools/plan/plan.goal

```
1  init module {
2    knowledge {
3      block(a).
4      block(b).
5      block(c).
6      clear(table).
7    }
8
9    beliefs {
10     on(c,a).
11     on(a,table).
12     on(b,table).
13     clear(c).
14     clear(b).
15   }
16
17   goals {
18     on(a,b), on(b,c).
19   }
20
21   actionspec {
22     move1(X,Y,Z)@int {
23       pre { block(X), block(Z), on(X,Y), clear(X), clear(Z) }
24       post { on(X,Z), not( on(X,Y) ), clear(Y), not( clear(Z) ) }
25     }
26
```

```
    move2(X,Y)@int {
      pre { block(X), on(X,Y), clear(X) }
      post { on(X,table), not( on(X,Y) ), clear(Y) }
    }
  }
}

main module [exit=nogoals] {
  program [order=plan] {
    % dummy action rule
    if true then print(0).
  }
}
```

src/test/java/goal/tools/plan/PlannerTest.java

```
package goal.tools.plan;

import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertNull;
import static org.junit.Assert.assertTrue;
import goal.core.agent.Agent;
import goal.core.agent.AgentId;
import goal.core.agent.EnvironmentCapabilities;
import goal.core.agent.GOALInterpreter;
import goal.core.agent.LoggingCapabilities;
import goal.core.agent.MessagingCapabilities;
import goal.core.agent.NoEnvironmentCapabilities;
import goal.core.agent.NoLoggingCapabilities;
import goal.core.agent.NoMessagingCapabilities;
import goal.core.kr.KRlanguage;
import goal.core.program.GOALProgram;
import goal.tools.PlatformManager;
import goal.tools.adapt.FileLearner;
import goal.tools.adapt.Learner;
import goal.tools.debugger.NOPDebugger;
import goal.tools.logging.Loggers;

import java.io.File;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import swiprolog3.engines.SWIPrologLanguage;

public class PlannerTest {

  @BeforeClass
  public static void setupBeforeClass() {
    Loggers.addConsoleLogger();
  }

  @AfterClass
  public static void tearDownAfterClass() {
    Loggers.removeConsoleLogger();
  }

  Agent<GOALInterpreter<NOPDebugger>> agent;
  GOALInterpreter<NOPDebugger> controller;
  KRlanguage language;
```

```
48
49    @Before
50    public void setUp() throws Exception {
51      AgentId id = new AgentId("TestAgent");
52      language = SWIPrologLanguage.getInstance();
53      File file = new File("src/test/resources/goal/tools/plan/plan.goal");
54      GOALProgram program = PlatformManager.parseGOALFile(file, language);
55      MessagingCapabilities messagingCpabilities = new NoMessagingCapabilities
        ();
56      EnvironmentCapabilities environmentCapabilities = new
        NoEnvironmentCapabilities();
57      LoggingCapabilities loggingCapabilities = new NoLoggingCapabilities();
58
59      NOPDebugger debugger = new NOPDebugger(id);
60      Learner learner = new FileLearner(id.getName(), program);
61      Planner planner = new POPPlanner(program);
62      controller = new GOALInterpreter<NOPDebugger>(program, debugger,
63          learner, planner);
64      agent = new Agent<GOALInterpreter<NOPDebugger>>(id,
65          environmentCapabilities, messagingCpabilities,
66          loggingCapabilities, controller);
67    }
68
69    @After
70    public void tearDown() throws Exception {
71      language.reset();
72    }
73
74    @Test
75    public void testStart() throws InterruptedException {
76      controller.start();
77      assertTrue(controller.isRunning());
78      controller.awaitTermination();
79      assertFalse(controller.isRunning());
80      assertNull(controller.getUncaughtThrowable());
81    }
82
83  }
```

src/main/java/goal/core/program/rules/RuleSet.java
(Everything but the `run` method of `RuleSet` is omitted)

```
303   /**
304    * Executes this {@link RuleSet}.
305    *
306    * @param runState
307    *            The run state in which the rule set is executed.
308    * @param substitution
309    *            The substitution provided by the module context that is
       passed
310    *            on to this rule set.
311    * @return The {@link Result} of executing this rule set.
312    * @throws KRQueryFailedException
313    *
314    *             FIXME: enable learner to deal with Rule#isSingleGoal
315    */
316   public Result run(RunState<?> runState, Substitution substitution) {
317     Result result = new Result();
318     // Make a copy of the rules so we don't shuffle the original below.
319     ArrayList<Rule> rules = new ArrayList<Rule>(this.rules);
320     MentalState ms;
321
322     switch (ruleOrder) {
```

```
323       case PLAN:
324         ms = runState.getMentalState();
325
326         Set<SingleGoal> goals = ms.getAttentionSet().getGoals();
327         if (goals.isEmpty()) {
328           // No goal to plan for
329           break;
330         }
331
332         // Search for plan
333         SingleGoal goal = goals.iterator().next();
334         List<Action> plan = runState.getPlanner().plan(ms, goal);
335
336         if (plan == null) {
337           // No plan found
338           break;
339         }
340
341         // Execute plan step by step
342         for (Action action : plan) {
343           result = action.run(runState, runState.getMentalState()
344               .getKRLanguage().getEmptySubstitution(),
345               runState.getDebugger());
346
347           if (result.hasPerformedAction()) {
348             // Update beliefs
349             runState.startCycle(true);
350           } else {
351             // Exit module if precondition fails
352             break;
353           }
354         }
355
356         break;
357       case ADAPTIVE:
358       case LINEARADAPTIVE:
359         /*
360          * For now there is no differentiation between adaptive and linear
361          * adaptive options. In both cases, a 'random' action option will be
362          * selected for execution by the learner.
363          */
364
365         ms = runState.getMentalState();
366         RuleSet ruleSet = this.applySubst(substitution);
367
368         runState.incrementRoundCounter();
369         runState.getDebugger().breakpoint(
370             Channel.REASONING_CYCLE_SEPARATOR,
371             null,
372             "+++++++ Adaptive Cycle " + runState.getRoundCounter()
373                 + " +++++++ ");
374
375         /*
376          * Get the learner to choose one action option, from the input list
377          * of action options.
378          */
379         List<ActionCombo> options = ruleSet.getActionOptions(ms,
380             runState.getDebugger());
381
382         // There are no possible options for actions to execute.
383         if (options.isEmpty()) {
384           break;
385         }
386
387         // Select an action
```

```
388         ActionCombo chosen = runState.getLearner().act(
389             runState.getActiveModule().getName(), ms, options);
390
391         /* Now execute the action option */
392         result = chosen.run(runState, substitution);
393
394         /*
395          * Obtain the reward from the environment. Or, if the environment
396          * does not support rewards, then create an internal reward based on
397          * whether we have achieved all our goals or not.
398          */
399         boolean goalsEmpty = ms.getAttentionSet().getGoals().isEmpty();
400         // runState should now have reward set.
401         Double envReward = runState.getReward();
402         double reward = (envReward != null) ? envReward : goalsEmpty ? 1.0
403             : 0.0;
404
405         if (!goalsEmpty) {
406           /* Update the learner with the reward from the last action */
407           runState.getLearner().update(
408               runState.getActiveModule().getName(), ms, reward);
409         } else {
410           /*
411            * If goals were achieved, then the final reward is calculated,
412            * and the learning episode finished, in RunState.kill() when
413            * the agent is killed.
414            */
415         }
416         break;
417       case RANDOM:
418         Collections.shuffle(rules);
419       case LINEAR:
420         for (Rule rule : rules) {
421           result = rule.run(runState, substitution);
422           if (result.isFinished()) {
423             break;
424           }
425         }
426         break;
427       case RANDOMALL:
428         Collections.shuffle(rules);
429       case LINEARALL:
430         // Continue evaluating and applying rule as long as there are more,
431         // and no {@link ExitModuleAction} has been performed.
432         for (Rule rule : rules) {
433           result.merge(rule.run(runState, substitution));
434           if (result.isModuleTerminated()) {
435             break;
436           }
437         }
438         break;
439       }
440
441       return result;
442     }
```

src/main/java/goal/tools/plan/Link.java

```
1  package goal.tools.plan;
2
3  import goal.core.kr.language.DatabaseFormula;
4
5  public class Link {
```

```
6
7     PlanAction provider, consumer;
8     DatabaseFormula proposition;
9
10    public Link(PlanAction provider, DatabaseFormula proposition,
11        PlanAction consumer) {
12      this.provider = provider;
13      this.proposition = proposition;
14      this.consumer = consumer;
15    }
16
17    @Override
18    public String toString() {
19      return "Link(" + provider + ",\n" + proposition + "," + consumer
20          + "\n)\n";
21    }
22
23  }
```

src/main/java/goal/tools/plan/Ordering.java

```
1   package goal.tools.plan;
2
3   public class Ordering {
4
5     private final PlanAction before, after;
6
7     public Ordering(PlanAction before, PlanAction after) {
8       this.before = before;
9       this.after = after;
10    }
11
12  }
```

src/main/java/goal/tools/plan/OrderingManager.java

```
1   package goal.tools.plan;
2
3   import java.util.Collections;
4   import java.util.HashMap;
5   import java.util.HashSet;
6   import java.util.IdentityHashMap;
7   import java.util.LinkedList;
8   import java.util.List;
9   import java.util.Map;
10  import java.util.Queue;
11  import java.util.Set;
12
13  public class OrderingManager {
14
15    private final Map<PlanAction, Set<PlanAction>> orderings = new
      IdentityHashMap<PlanAction, Set<PlanAction>>();
16    private final Map<PlanAction, Set<PlanAction>> closure = new
      IdentityHashMap<PlanAction, Set<PlanAction>>();
17
18    /**
19     * Check whether or not an ordering constraint is consistent with the rest
       .
20     *
21     * @param a
22     *            plan action that comes first.
```

```
23      * @param b
24      *            plan action that comes after.
25      * @return <code>true</code> iff the ordering constraint is consistent
        with
26      *         the rest.
27      */
28     public boolean isConsistent(PlanAction a, PlanAction b) {
29       // Check for cycles
30       Set<PlanAction> afterB = closure.get(b);
31       return afterB == null || !afterB.contains(a);
32     }
33
34     /**
35      * Get all plan actions that come after a specific plan action.
36      *
37      * @param a
38      *            plan action.
39      * @return A set of actions that come after the action specified.
40      */
41     public Set<PlanAction> after(PlanAction a) {
42       Set<PlanAction> after = closure.get(a);
43       if (after == null) {
44         return Collections.EMPTY_SET;
45       }
46       return after;
47     }
48
49     // TODO: Leave consistency check to caller?
50     /**
51      * Add an ordering constraint if it's consistent with the rest.
52      *
53      * @param a
54      *            plan action that comes first.
55      * @param b
56      *            plan action that comes after.
57      *
58      * @return <code>true</code> iff ordering constraint was added
        successfully.
59      */
60     public boolean addIfConsistent(PlanAction a, PlanAction b) {
61       if (!isConsistent(a, b)) {
62         return false;
63       }
64
65       Set<PlanAction> afterA = orderings.get(a);
66       Set<PlanAction> afterAClosed;
67
68       if (afterA == null) {
69         afterA = new HashSet<PlanAction>();
70         orderings.put(a, afterA);
71         afterAClosed = new HashSet<PlanAction>();
72         closure.put(a, afterAClosed);
73       } else {
74         afterAClosed = closure.get(a);
75       }
76
77       Set<PlanAction> afterBClosed = closure.get(b);
78       if (afterBClosed == null) {
79         afterBClosed = Collections.EMPTY_SET;
80       }
81
82       afterA.add(b);
83       afterAClosed.add(b);
84       afterAClosed.addAll(afterBClosed);
85
```

```
86      for (PlanAction action : orderings.keySet()) {
87        Set<PlanAction> afterClosed = closure.get(action);
88        if (afterClosed != null && afterClosed.contains(a)) {
89          afterClosed.add(b);
90          afterClosed.addAll(afterBClosed);
91        }
92      }
93
94      return true;
95    }
96
97    /**
98     * Remove an ordering constraint. Note that this will trigger a
99     * re-propagation of the transitive closure.
100    *
101    * @param a
102    *            plan action that comes first.
103    * @param b
104    *            plan action that comes after.
105    * @return <code>true</code> iff this update had any effect.
106    */
107   public boolean remove(PlanAction a, PlanAction b) {
108     Set<PlanAction> afterA = orderings.get(a);
109     if (afterA == null || !afterA.contains(b)) {
110       return false;
111     }
112
113     // Compute linearization before removal
114     List<PlanAction> linearization = getLinearization();
115
116     // Remove ordering constraint
117     afterA.remove(b);
118     if (afterA.isEmpty()) {
119       orderings.remove(a);
120     }
121
122     // Re-propagate transitive closure
123     closure.clear();
124
125     for (int i = linearization.size() - 1; i >= 0; i--) {
126       PlanAction x = linearization.get(i);
127       Set<PlanAction> afterXClosed = closure.get(x);
128
129       for (int j = i - 1; j >= 0; j--) {
130         PlanAction y = linearization.get(j);
131
132         Set<PlanAction> afterY = orderings.get(y);
133         if (afterY != null && afterY.contains(x)) {
134           Set<PlanAction> afterYClosed = closure.get(y);
135           if (afterYClosed == null) {
136             afterYClosed = new HashSet<PlanAction>();
137             closure.put(y, afterYClosed);
138           }
139
140           afterYClosed.add(x);
141           if (afterXClosed != null) {
142             afterYClosed.addAll(afterXClosed);
143           }
144         }
145       }
146     }
147
148     return true;
149   }
150
```

```
151   /**
152    * Compute a topological sort of the ordering constraints.
153    *
154    * @return A total-order plan that satisfies all ordering constraints.
155    */
156   public List<PlanAction> getLinearization() {
157     List<PlanAction> linearization = new LinkedList<PlanAction>();
158
159     // Make copy of orderings
160     Map<PlanAction, Set<PlanAction>> edges = new HashMap<PlanAction, Set<
      PlanAction>>(
161         orderings);
162
163     // Find nodes with no incoming edges
164     Queue<PlanAction> noIncoming = new LinkedList<PlanAction>(
165         orderings.keySet());
166     for (Set<PlanAction> actions : orderings.values()) {
167       noIncoming.removeAll(actions);
168     }
169
170     while (!noIncoming.isEmpty()) {
171       PlanAction n = noIncoming.poll();
172       linearization.add(n);
173
174       Set<PlanAction> afterN = edges.get(n);
175       if (afterN == null) {
176         continue;
177       }
178
179       edges.remove(n);
180
181       forAfterN: for (PlanAction m : afterN) {
182         for (Set<PlanAction> actions : edges.values()) {
183           if (actions.contains(m)) {
184             continue forAfterN;
185           }
186         }
187
188         noIncoming.add(m);
189       }
190
191     }
192
193     return linearization;
194   }
195 }
```

src/main/java/goal/tools/plan/POPPlanner.java

```
1  package goal.tools.plan;
2
3  import goal.core.kr.language.DatabaseFormula;
4  import goal.core.kr.language.Substitution;
5  import goal.core.mentalstate.BASETYPE;
6  import goal.core.mentalstate.BeliefBase;
7  import goal.core.mentalstate.MentalState;
8  import goal.core.mentalstate.SingleGoal;
9  import goal.core.program.ActionSpecification;
10 import goal.core.program.GOALProgram;
11 import goal.core.program.actions.Action;
12
13 import java.util.ArrayList;
14 import java.util.Collections;
```

```
15 import java.util.HashSet;
16 import java.util.LinkedList;
17 import java.util.List;
18 import java.util.Set;
19
20 public class POPPlanner implements Planner {
21
22   private class AgendaPair {
23
24     public PlanAction action;
25     public DatabaseFormula precondition;
26
27     public AgendaPair(PlanAction action, DatabaseFormula precondition) {
28       this.action = action;
29       this.precondition = precondition;
30     }
31
32     @Override
33     public String toString() {
34       return "AgendaPair(" + action.toString() + ","
35           + precondition.toString() + ")";
36     }
37
38   }
39
40   private class Provider {
41
42     public PlanAction action;
43     public Substitution subst;
44
45     public Provider(PlanAction action, Substitution subst) {
46       this.action = action;
47       this.subst = subst;
48     }
49
50     @Override
51     public String toString() {
52       return "ProviderPair(" + action.toString() + "," + subst.toString()
53           + ")";
54     }
55   }
56
57   private final Substitution emptySubst;
58   private final List<ActionSpecification> actionSpecs;
59   private long nonce;
60
61   /**
62    * Create a new partial-order planner.
63    *
64    * @param program
65    *            the goal program
66    */
67   public POPPlanner(GOALProgram program) {
68     emptySubst = program.getKRLanguage().getEmptySubstitution();
69     actionSpecs = program.getModule("init").getActionSpecifications();
70   }
71
72   @Override
73   public List<Action> plan(MentalState ms, SingleGoal goal) {
74     nonce = 0;
75
76     Set<DatabaseFormula> startActionPositiveEffects = new HashSet<
       DatabaseFormula>();
77     BeliefBase bb = ms.getOwnBase(BASETYPE.BELIEFBASE);
78     BeliefBase kb = ms.getOwnBase(BASETYPE.KNOWLEDGEBASE);
```

```
79      startActionPositiveEffects.addAll(bb.getTheory().getFormulas());
80      startActionPositiveEffects.addAll(kb.getTheory().getFormulas());
81
82      Set<DatabaseFormula> finishActionPreconditions = new HashSet<
        DatabaseFormula>();
83      finishActionPreconditions.addAll(goal.getGoal().getAddList());
84
85      PlanAction startAction = new PlanAction(null, Collections.EMPTY_SET,
86          startActionPositiveEffects, Collections.EMPTY_SET);
87      PlanAction finishAction = new PlanAction(null,
88          finishActionPreconditions, Collections.EMPTY_SET,
89          Collections.EMPTY_SET);
90
91      Plan plan = new Plan(startAction, finishAction, emptySubst);
92      LinkedList<AgendaPair> agenda = new LinkedList<AgendaPair>();
93
94      for (DatabaseFormula precondition : finishActionPreconditions) {
95        agenda.add(new AgendaPair(finishAction, precondition));
96      }
97
98      plan = pop(plan, agenda);
99      if (plan != null) {
100       return plan.getLinearization();
101     }
102
103     // No plan found
104     return null;
105   }
106
107   // Will mutate plan!
108   private Plan pop(Plan plan, LinkedList<AgendaPair> agenda) {
109     if (agenda.isEmpty()) {
110       // Nothing to do, all sub-goals satisfied
111       return plan;
112     }
113
114     // Next sub-goal to satisfy
115     AgendaPair pair = agenda.poll();
116
117     List<Provider> providers = getProviders(pair, plan);
118
119     // Non-deterministic choice of relevant action
120     for (Provider provider : providers) {
121       PlanAction action = provider.action;
122       Substitution subst = provider.subst;
123
124       // Update binding constraints
125       Substitution oldSubst = plan.subst;
126       Substitution newSubst = plan.subst.combine(subst);
127       if (newSubst == null) {
128         continue;
129       }
130       plan.subst = newSubst;
131
132       // Add causal link
133       Link link = new Link(action, pair.precondition, pair.action);
134       plan.links.add(link);
135
136       boolean isNewAction = !plan.actions.contains(action);
137       if (isNewAction) {
138         plan.actions.add(action);
139
140         plan.orderingManager.addIfConsistent(plan.start, action);
141         plan.orderingManager.addIfConsistent(action, plan.finish);
142
```

```
143         for (DatabaseFormula precondition : action.getPreconditions()) {
144           agenda.push(new AgendaPair(action, precondition));
145         }
146       }
147
148       if (plan.orderingManager.addIfConsistent(action, pair.action)) {
149         LinkedList<Threat> threats = getThreats(plan, isNewAction,
150             action, link);
151
152         Plan result = pop2(plan, agenda, threats);
153         if (result != null) {
154           return result;
155         }
156
157         // TODO: These removals will all trigger re-propagation!
158         plan.orderingManager.remove(action, pair.action);
159       }
160
161       // Remove new action
162       if (isNewAction) {
163         for (int i = 0; i < action.getPreconditions().size(); i++) {
164           agenda.pop();
165         }
166
167         // TODO: These removals will all trigger re-propagation!
168         plan.orderingManager.remove(action, plan.finish);
169         plan.orderingManager.remove(plan.start, action);
170
171         plan.actions.remove(action);
172       }
173
174       // Remove causal link
175       plan.links.remove(link);
176
177       // Revert binding constraints
178       plan.subst = oldSubst;
179     }
180
181     // Fail: Relevant actions exhausted
182     return null;
183   }
184
185   private Plan pop2(Plan plan, LinkedList<AgendaPair> agenda,
186       LinkedList<Threat> threats) {
187     if (threats.isEmpty()) {
188       return pop(plan, (LinkedList<AgendaPair>) agenda.clone());
189     }
190
191     Threat threat = threats.pop();
192     List<Resolver> resolvers = threat.getResolvers(plan);
193
194     // Resolvers: promote, demote, binding constraint?
195     for (Resolver resolver : resolvers) {
196       if (!resolver.apply()) {
197         continue;
198       }
199
200       Plan result = pop2(plan, agenda,
201           (LinkedList<Threat>) threats.clone());
202       if (result != null) {
203         return result;
204       }
205
206       resolver.revert();
207     }
```

```
208
209       // Fail: Resolvers exhausted
210       return null;
211     }
212
213     private List<Provider> getProviders(AgendaPair pair, Plan plan) {
214       List<Provider> providers = new ArrayList<Provider>();
215
216       // TODO: Multiple providers for same action?
217
218       // Add relevant existing actions
219       for (PlanAction action : plan.actions) {
220         // TODO: Ensure action can be ordered before pair.action? Does this
221         // fix it?
222         if (action == pair.action
223             || plan.orderingManager.after(pair.action).contains(action)) {
224           continue;
225         }
226
227         for (DatabaseFormula formula : action.getPositiveEffects()) {
228           Substitution subst = formula.mgu(pair.precondition);
229
230           // TODO: Apply plan.subst to operands before mgu?
231           // Ensure the substitution is consistent with existing bindings
232           if (subst != null && plan.subst.combine(subst) != null) {
233             providers.add(new Provider(action, subst));
234             break;
235           }
236         }
237       }
238
239       // Add relevant new actions
240       for (ActionSpecification actionSpec : actionSpecs) {
241         PlanAction action = new PlanAction(actionSpec, nonce++, emptySubst);
242         for (DatabaseFormula formula : action.getPositiveEffects()) {
243           Substitution subst = formula.mgu(pair.precondition);
244           if (subst != null) {
245             providers.add(new Provider(action, subst));
246             break;
247           }
248         }
249       }
250
251       return providers;
252     }
253
254     private static boolean isThreat(Plan plan, PlanAction action, Link link) {
255       if (!plan.orderingManager.isConsistent(link.provider, action)
256           || !plan.orderingManager.isConsistent(action, link.consumer)) {
257         return false;
258       }
259
260       for (DatabaseFormula q : action.getNegativeEffects()) {
261         Substitution subst = link.proposition.mgu(q);
262         if (subst != null && plan.subst.combine(subst) != null) {
263           return true;
264         }
265       }
266
267       return false;
268     }
269
270     private static LinkedList<Threat> getThreats(Plan plan,
271         boolean isNewAction, PlanAction newAction, Link newLink) {
272       LinkedList<Threat> threats = new LinkedList<Threat>();
```

```
273
274     // Find threats due to new action
275     if (isNewAction) {
276       for (Link link : plan.links) {
277         if (link.provider == newAction || link.consumer == newAction) {
278           continue;
279         }
280
281         if (isThreat(plan, newAction, link)) {
282           threats.add(new Threat(newAction, link));
283         }
284       }
285     }
286
287     // Find threats on new link
288     for (PlanAction action : plan.actions) {
289       if (action == newLink.provider || action == newLink.consumer) {
290         continue;
291       }
292
293       if (isThreat(plan, action, newLink)) {
294         threats.add(new Threat(action, newLink));
295       }
296     }
297
298     return threats;
299   }
300 }
```

src/main/java/goal/tools/plan/Plan.java

```
1  package goal.tools.plan;
2
3  import goal.core.kr.language.Substitution;
4  import goal.core.program.actions.Action;
5
6  import java.util.ArrayList;
7  import java.util.HashSet;
8  import java.util.List;
9  import java.util.Set;
10
11 /**
12  * A partially instantiated partial-order plan.
13  *
14  * @author S.Jacobi
15  *
16  */
17 public class Plan {
18
19   public final PlanAction start, finish;
20   public final Set<PlanAction> actions = new HashSet<PlanAction>();
21   public final OrderingManager orderingManager = new OrderingManager();
22   public Substitution subst;
23   public final Set<Link> links = new HashSet<Link>();
24
25   /**
26    * Create new partial-order plan with pseudo actions start and finish.
27    *
28    * @param start
29    *            plan action with initial state as positive effects.
30    * @param finish
31    *            plan action with goal state as preconditions.
32    * @param emptySubst
```

```
33      */
34    public Plan(PlanAction start, PlanAction finish, Substitution emptySubst)
      {
35      this.start = start;
36      this.finish = finish;
37      actions.add(start);
38      actions.add(finish);
39      orderingManager.addIfConsistent(start, finish);
40      subst = emptySubst;
41    }
42
43    /**
44     * Get one possible fully instantiated linearization of the partial-order
45     * plan.
46     *
47     * @return A list of {@link Action}s that describe a possible total-order.
48     */
49    public List<Action> getLinearization() {
50      // TODO: What if flawless plan is partially instantiated?
51
52      List<PlanAction> planActionLinearization = orderingManager
53          .getLinearization();
54      List<Action> linearization = new ArrayList<Action>(
55          planActionLinearization.size() - 2);
56
57      for (PlanAction planAction : planActionLinearization) {
58        Action action = planAction.getAction();
59        if (action != null) {
60          linearization.add(action.applySubst(subst));
61        }
62      }
63
64      return linearization;
65    }
66
67  }
```

src/main/java/goal/tools/plan/PlanAction.java

```
1   package goal.tools.plan;
2
3   import goal.core.kr.language.DatabaseFormula;
4   import goal.core.kr.language.Substitution;
5   import goal.core.kr.language.Var;
6   import goal.core.program.ActionSpecification;
7   import goal.core.program.actions.Action;
8   import goal.core.program.actions.UserSpecAction;
9   import goal.tools.errorhandling.exceptions.KRInitFailedException;
10
11  import java.util.Collections;
12  import java.util.HashSet;
13  import java.util.List;
14  import java.util.Set;
15
16  public class PlanAction {
17
18    private static final String VARIABLE_POSTFIX = "planactionpostfix";
19
20    private final UserSpecAction action;
21    private final Set<DatabaseFormula> preconditions;
22    private final Set<DatabaseFormula> positiveEffects;
23    private final Set<DatabaseFormula> negativeEffects;
24
```

```
25    /**
26     * Create new plan action from action specification and unique nonce.
27     *
28     * @param actionSpec
29     *            action specification to create plan action from
30     * @param nonce
31     *            number unique to this call within current plan search
32     * @param emptySubst
33     *            empty substitution for relevant kr language
34     */
35    public PlanAction(ActionSpecification actionSpec, long nonce,
36        Substitution emptySubst) {
37      preconditions = new HashSet<DatabaseFormula>();
38      positiveEffects = new HashSet<DatabaseFormula>();
39      negativeEffects = new HashSet<DatabaseFormula>();
40
41      List<DatabaseFormula> preconditionList = actionSpec.getPreCondition()
42          .toUpdate().getAddList();
43      List<DatabaseFormula> positiveEffectsList = actionSpec
44          .getPostCondition().getAddList();
45      List<DatabaseFormula> negativeEffectsList = actionSpec
46          .getPostCondition().getDeleteList();
47
48      // Rename variables
49      Substitution renameSubst = emptySubst;
50      for (DatabaseFormula formula : preconditionList) {
51        renameSubst = updateRenameSubst(renameSubst, formula, nonce);
52        preconditions.add(formula.applySubst(renameSubst));
53      }
54      for (DatabaseFormula formula : positiveEffectsList) {
55        renameSubst = updateRenameSubst(renameSubst, formula, nonce);
56        positiveEffects.add(formula.applySubst(renameSubst));
57      }
58      for (DatabaseFormula formula : negativeEffectsList) {
59        renameSubst = updateRenameSubst(renameSubst, formula, nonce);
60        negativeEffects.add(formula.applySubst(renameSubst));
61      }
62
63      action = actionSpec.getAction().applySubst(renameSubst);
64
65      try {
66        action.addSpecification(actionSpec);
67      } catch (KRInitFailedException e) {
68        e.printStackTrace();
69      }
70    }
71
72    public PlanAction(UserSpecAction action,
73        Set<DatabaseFormula> preconditions,
74        Set<DatabaseFormula> positiveEffects,
75        Set<DatabaseFormula> negativeEffects) {
76      this.action = action;
77      this.preconditions = preconditions;
78      this.positiveEffects = positiveEffects;
79      this.negativeEffects = negativeEffects;
80    }
81
82    private static Substitution updateRenameSubst(Substitution subst,
83        DatabaseFormula formula, long nonce) {
84      for (Var var : formula.getFreeVar()) {
85        if (!subst.getVariables().contains(var)) {
86          subst = subst.combine(var.renameVar("", VARIABLE_POSTFIX
87              + nonce));
88        }
89      }
```

```
90
91      return subst;
92    }
93
94    public Action getAction() {
95      return action;
96    }
97
98    public Set<DatabaseFormula> getPreconditions() {
99      return Collections.unmodifiableSet(preconditions);
100   }
101
102   public Set<DatabaseFormula> getPositiveEffects() {
103     return Collections.unmodifiableSet(positiveEffects);
104   }
105
106   public Set<DatabaseFormula> getNegativeEffects() {
107     return Collections.unmodifiableSet(negativeEffects);
108   }
109
110   @Override
111   public String toString() {
112     StringBuilder sb = new StringBuilder();
113     sb.append("\n<");
114     sb.append(action);
115     sb.append(",\npre: ");
116     sb.append(preconditions);
117     sb.append(",\nef+: ");
118     sb.append(positiveEffects);
119     sb.append(",\nef-: ");
120     sb.append(negativeEffects);
121     sb.append(">");
122
123     return sb.toString();
124   }
125
126 }
```

src/main/java/goal/tools/plan/Planner.java

```
1  package goal.tools.plan;
2
3  import goal.core.mentalstate.MentalState;
4  import goal.core.mentalstate.SingleGoal;
5  import goal.core.program.actions.Action;
6
7  import java.util.List;
8
9  public interface Planner {
10
11   /**
12    * Search for a plan to achieve a single goal.
13    *
14    * @param ms
15    *            the current mental state
16    * @param goal
17    *            the goal to plan for
18    * @return A list of {@link Action}s that describes a plan to achieve the
19    *         goal or {@code null} if no plan is found.
20    */
21   List<Action> plan(MentalState ms, SingleGoal goal);
22
23 }
```

src/main/java/goal/tools/plan/Resolver.java

```
package goal.tools.plan;

public interface Resolver {

  public boolean apply();

  public void revert();

}
```

src/main/java/goal/tools/plan/ResolverDemote.java

```
package goal.tools.plan;

public class ResolverDemote implements Resolver {

  private final Plan plan;
  private final Threat threat;

  public ResolverDemote(Plan plan, Threat threat) {
    this.plan = plan;
    this.threat = threat;
  }

  @Override
  public boolean apply() {
    return plan.orderingManager.addIfConsistent(threat.link.consumer,
        threat.action);
  }

  @Override
  public void revert() {
    plan.orderingManager.remove(threat.link.consumer, threat.action);
  }

}
```

src/main/java/goal/tools/plan/ResolverPromote.java

```
package goal.tools.plan;

public class ResolverPromote implements Resolver {

  private final Plan plan;
  private final Threat threat;

  public ResolverPromote(Plan plan, Threat threat) {
    this.plan = plan;
    this.threat = threat;
  }

  @Override
  public boolean apply() {
    return plan.orderingManager.addIfConsistent(threat.action,
        threat.link.provider);
  }

  @Override
  public void revert() {
    plan.orderingManager.remove(threat.action, threat.link.provider);
```

```
22    }
23
24  }
```

src/main/java/goal/tools/plan/Threat.java

```
1   package goal.tools.plan;
2
3   import java.util.ArrayList;
4   import java.util.List;
5
6   public class Threat {
7
8     public PlanAction action;
9     public Link link;
10
11    public Threat(PlanAction action, Link link) {
12      this.action = action;
13      this.link = link;
14    }
15
16    public List<Resolver> getResolvers(Plan plan) {
17      List<Resolver> resolvers = new ArrayList<Resolver>();
18
19      resolvers.add(new ResolverDemote(plan, this));
20      resolvers.add(new ResolverPromote(plan, this));
21      // TODO: Resolve threat with binding constraint?
22
23      return resolvers;
24    }
25  }
```

Bibliography

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[Hin14] Koen V. Hindriks. *Programming Cognitive Agents in GOAL*. Published at `http://ii.tudelft.nl/trac/goal/` (accessed January 2014), 2014.

[RN10] Stuart Russell and Peter Norvig. *Artificial intelligence: A Modern Approach*. Pearson, 2010.