An Alternative Approach for Reasoning about the Goal-Plan Tree Problem

Patricia Shaw[1] and Rafael H. Bordini[2]

[1] University of Durham, UK
p.h.shaw@durham.ac.uk
[2] Federal University of Rio Grande do Sul, Brazil
r.bordini@inf.ufrgs.br

Abstract. Agents programmed in BDI-inspired languages have goals to achieve and a library of plans that can be used to achieve them, typically requiring further goals to be adopted. This is most naturally represented by a structure that has been called a Goal-Plan Tree. One of the uses of such structure is in agent deliberation (in particular, deciding whether to commit to achieving a certain goal or not). In previous work, a Petri net based approach for reasoning about goal-plan trees was defined. This paper presents a constraint-based approach to perform the same reasoning, which is then compared with the Petri net approach.

Keywords: Agent Reasoning, Constraints, Goal-Plan Tree.

1 Introduction

Agents programmed in BDI-inspired languages have goals to achieve and a library of plans that can be used to achieve them, typically requiring further goals to be adopted. This is most naturally represented by a structure that has been called a Goal-Plan Tree. Whilst no planning takes place in such agents, a certain type of reasoning – done over such representation of agents' commitments towards goals to be achieved and the known courses of actions to achieve them – can significantly impact the agent's performance by judicious scheduling of the plan execution. More importantly, it can significantly improve *deliberation*, in the sense that an agent can make reasoned choices on whether to commit to achieving a new goal or not.

In the work by Thangarajah *et al.* [8,9,10], a *goal-plan tree* is used to represent the structure of the various plans and subgoals related to each goal for an individual agent. At each node of the tree, *summary information* is used to represent the various constraints under consideration. This is similar to previous work by Clement and Durfee [1,2,3], using summary information with Hierarchical Task Network (HTN) planning to co-ordinate the actions of multiple agents.

When using summary information, the amount of summary information to handle could potentially grow exponentially with the size of the goal-plan tree [3], which could have a significant impact on the performance of the agent for larger problems. A different approach was introduced by Shaw and Bordini [5], where

M. Dastani et al. (Eds.): LADS 2010, LNAI 6822, pp. 115–135, 2011.
© Springer-Verlag Berlin Heidelberg 2011

a goal-plan tree is mapped into a Petri net in such a way as to avoid the need for summary information.

The work in [5] considered reasoning about both positive and negative effects of a plan on other plans using a Petri net based technique, while in [6] the focus is on reasoning about resources using Petri nets, which are then *combined* into a coherent reasoning process encompassing the reasoning about positive and negative interactions from [5]. These were evaluated based on an abstract scenario as well as a more concrete scenario using a simplified mars rover scenario.

In this paper, we present an alternative specific implementation of an approach to reasoning about positive, negative and resource interactions using a constraint logic programming approach developed in GNU Prolog to define a set of constraints that are eventually solved to generate a successful execution ordering of the plans to achieve an agent's goals. These are evaluated against the Petri net model using a common abstract scenario. While the approach described here is based on a specific implementation, the concepts and processes could be reapplied in other constraint (logic) programming or even constraint optimisation settings. However, the aim of this paper is to present an approach to solving a problem and experimentally compare it to another approach in order to identify situations where it may be preferable to apply one approach over the other.

The remainder of the paper is organised as follows. Section 2 shows the constraint-based approach with each of the three forms of reasoning incorporated. Section 3 shows the experimental results and analysis of the comparison of that approach to the Petri net approach for reasoning about the goal-plan tree problem. Section 4 concludes the paper.

2 Constraint-Based Approach

2.1 Goal-Plan Trees

A goal-plan tree consists of a top-level goal at the root, with one or more plans available to achieve that goal. Each of these plans may themselves include further subgoals forming the next level in the tree, followed by additional plans to achieve these subgoals[1]. All subgoals for a plan must be achieved for a plan to be successful, while only one plan option needs to be executed for a goal or subgoal to be successful. The simplest plans at the leaves of the tree will just contain a sequence of actions and no further subgoals. An example of a goal-plan tree is shown in Figure 1, which shows the goal-plan tree representation of a goal for a Mars Rover to collect a soil sample from a location then transmit the results back to Earth via the base station.

An agent will most likely have multiple top-level goals to achieve, each with its own goal-plan tree. While it is often straightforward for these to be achieved in sequence, it may be possible for the agent to achieve better performance by

[1] The term subgoals will always be used when referring to subgoals, while top-level goals will either be referred to as goals or top-level goals.

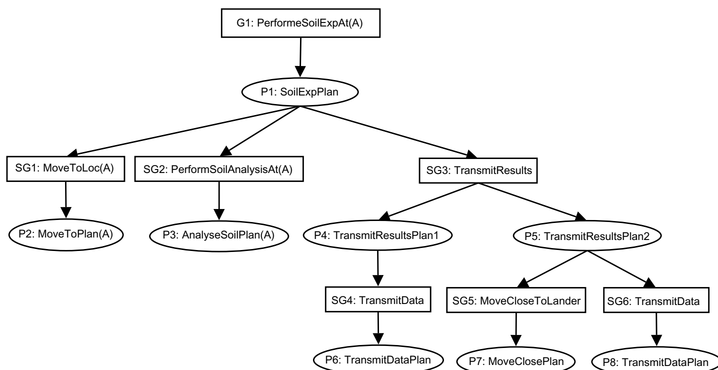


Fig. 1. Goal-plan tree for a Mars rover as used by Thangarajah *et al.* The goals and subgoals are represented by rectangles while the plans are represented by ovals.

attempting to achieve them in parallel. This can of course lead to problems where the goals interfere with each other and where resources are limited, so reasoning about that can help an agent succeed in achieving its goals and do so more efficiently. The three types of reasoning considered here are based on: (i) the limited availability of consumable resources, (ii) the potential for positive interactions between goals, and (iii) the risk of negative interference between goals, which are discussed in more detail in the following sections.

The approach developed here for reasoning about goals applies constraint satisfaction to find a solution to instances of the goal-plan tree problem. While the Petri net approach applied in [6] provided a natural representation of an agent's goal-plan tree into which the reasoning could be added, this approach provides a natural representation of the constraints to be handled by the agent in the form of resource constraints and interaction constraints. The constraints are represented using GNU Prolog[2].

2.2 Modelling a Goal-Plan Tree

The idea surrounding the model used for representing the goal-plan tree reasoning problem as a set of constraints is to find an ordering of the plans for all of the goals such that all the goals adopted are achieved and as many goals as possible are adopted.

To start with, the plans and goals are both defined as facts using a Prolog functor `node`, with the plans being represented by 5-tuples $\langle Pl, S, Pr, E, R\rangle$ where Pl is a unique identifier for each plan; S is the list of subgoals for achieving the plan; Pr is a list of preconditions and E is a list of effects caused by the plan; R is a list of pairs showing the resource requirements for the different resources that a plan uses. The plans at the bottom of the goal-plan tree that

[2] The shorthand notation `V1/V2` is used in GNU Prolog to represent pairs of values.

form the leaves of the tree will not have any subgoals listed in S, and not all plans will have preconditions, effects or resource requirements.

A series of "variables" is used to represent resources and the effects on the environment. The representation of available resources make use of dynamic facts that can be updated as the resources are consumed, (e.g. `resource(r1,50)`). In the plan definitions, the preconditions, effects and resources are all represented as pairs of values, for example `r1/5` represents the requirement of 5 units of resource $r1$. The preconditions and effects referring to various properties of the environment that can be modified are represented in a similar way with effect `e1/7` stating that the plan changes the variable representing the environment property $e1$ so that it has the value 7. Sections 2.4 and 2.5 describe how these are used to identify plans that can either be safely "merged" or that could interfere (thus needing to be scheduled accordingly).

Goals and subgoals require less details, so they are simply represented as pairs $\langle G, P\rangle$, where G is a unique identifier for the goal or subgoal and P is a non-empty list of plans that can be used to achieve G. The following Prolog sample from a goal definition shows a top-level goal node and a plan node that achieves this goal, itself using 1 unit of resource r1 and causing the effect of assigning the value 7 to variable e3, while having no preconditions required for it to start.

```
node(g1,[p1]).                              % Goal node
node(p1,[sg2,sg3],[],[e3/7],[r1/1]).   % Plan node
```

In order to reason about the tree structure, various predicates are defined to help query a goal-plan tree representation. These include listing all the plans in the sub-tree of a goal or plan, finding all the plan options for achieving a goal or subgoal, and querying the plan hierarchy within the goal-plan tree.

Where there is a choice of plans to achieve a goal or subgoal, only one of these needs to be used in order for the goal to be successful. The surplus plans can therefore be dropped from consideration, reducing the number of plans that need to be considered later on. When the plan being dropped contains subgoals, these are also removed from consideration. This is illustrated in Figure 2, where the plan and its sub-tree inside the dashed line is being dropped in preference of the alternative plan for achieving the subgoal.

In Prolog, this is defined as a series of predicates to "strip" the tree of the branch options:

```
branchOptions:-
    findall(O,option(_,O),All),
    branchStrip(All).
```

The clause above uses the `option(Goal,OptionList)` predicate to generate a list of all the sets of options for subgoal branches. `O` is a list of plans from which just one plan needs to be selected, so the variable `All`, the result of the `findall`, equates to a list of plan lists. Each of these lists of plans then needs to be considered, selecting one plan to keep and the remainder to disregard. By default, the plan that is kept is the first plan in the list. However, when resource

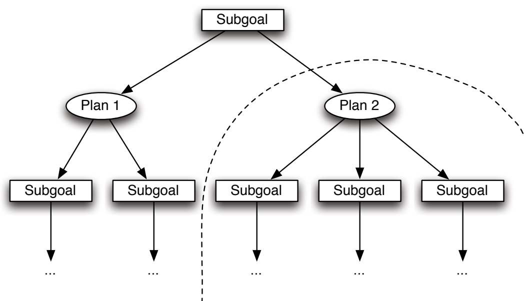


Fig. 2. Removal of surplus sub-trees where there is a choice of plans

reasoning is incorporated, the summary resource requirements for each branch is considered so the plan with the lowest summary resource requirements is kept.

```
branchStrip([]).
branchStrip([[H | T2] | T]):-
    rmBranch(T2),
    branchStrip(T).

rmBranch([]).
rmBranch([P|T]):-
    strip(P),
    rmBranch(T).
```

When removing plans, it is important to remember to remove the sub-tree formed from any subgoals that were required by the plan. This is handled by a final recursive predicate to iterate through the list ensuring each of the members of the sub-tree are removed. As it is possible for plans within the subtree of an optional plan to also contain branches, it is possible for plans and subgoals to have already been removed. To prevent this from causing the retraction to fail a disjunction finishing with `true` is included as shown below.

```
strip(P):-
    subtree(P,T),!,
    stripTree(T),
    retract(node(P,_,_,_,_)).

stripTree([]).
stripTree([H|T]):-
    (((retract(node(H,_,_,_,_))); retract(node(H,_))); true),
    stripTree(T).
```

An evaluation of the Constraint Satisfaction Problem (CSP) gives each plan that is considered a number that can be used to sequence the plans. A global

finite-domain variable is created for each of the plans to store a value in the domain of plans, ranging from 0 to the number of plans. A solution is a valid sequence where the goals adopted would be achieved if the plans were executed in the order specified by the evaluation. A tree scheduling predicate, `treeScheduler` defined below, is applied to the plan variables to ensure the tree structure is maintained when considering the order in which to execute plans, forming the basis of any scheduling over the plans. This includes preconditions and effects of plans between different branches within a tree to ensure a plan is not scheduled to execute before the plan producing the necessary preconditions has been scheduled to execute.

```
treeScheduler([]).
treeScheduler([[P1,P2]|T]):-
    g_read(P1,I),
    g_read(P2,J),
    I#<#J,
    g_assign(P1,I),
    g_assign(P2,J),
    treeScheduler(T).
```

In many cases, the ordering between subsets of the plans is not important as they will not affect each other in any way, so these plans can safely be given the same sequence number. When executing the plans, this could be seen as either executing them in parallel or executing them in sets, such that all the plans with sequence number 1 are executed before those with sequence number 2, and so forth. By not specifying an exact ordering of the plans, the agent is able to maintain a lot of its autonomy when selecting which plan to execute next. Essentially the "ordering" of plans indicates to the agent which plans are safe to execute together, grouping them into "safe" sets. Provided the agent completes all the plans within one group before moving on to the next, there should be no interference between the various goals. In the worst case, where there was a lot of interference between all of the goals, each plan could be assigned a unique number from their domain of values, specifying the exact ordering in which the plans must be executed for the agent to be successful.

When searching for valid solutions to the goal-plan tree problem, the query is directed from the `reasoning` predicate shown below. When a solution is found, each of the parameters in the head of the predicate is unified with part of the solution or details about the solution for evaluation purposes. This includes counting the number of plans used, the number of goals achieved and the time taken for the solution to be found. The Prolog predicate `real_time(Time)` is used to obtain start and end timings for the evaluation of a goal-plan tree model.

```
reasoning(Schedule, Plans, PlanCount, TimeTaken,
    GoalsSet, GoalsAchieved):-
    real_time(Start),        % start timing the reasoning

    findall(G,root(G),Goals),
```

```
length(Goals, GoalsSet),

branchOptions,
                             % positive interaction reasoning
findall([Pa,Pb],pos(Pa,Pb),Merge),
posScheduler(Merge),
                             % resource reasoning
branchList(Goals,SumList),
sort(SumList,SortedSumList),
resReasoning(SortedSumList),

findall(P,node(P,_,_,_,_),Plans),
length(Plans,PlanCount),
varSetup(Plans,PlanCount),

findall([Px,Py],tree(Px,Py),A),
reverse(A,A2),
treeScheduler(A2),
                             % negative interference reasoning
findall([Pc,Pd,Pe],neg(Pc,Pd,Pe),Neg),
negScheduler(Neg),

varResult(Plans,Schedule),
fd_labeling(Schedule,[variable_method(standard)]),

real_time(End),              % reasoning finished
TimeTaken#=End-Start,
findall(G2,root(G2),Goals2),
length(Goals2,GoalsAchieved).
```

The first step in the clause unifies the variable `Goals` with a list of all the top-level goals. The length of this list is queried to identify how many goals have been defined at the start. When reasoning about consumable resources, it is likely that not all goals will be achieved, so a repeat of this is performed to count the number of goals after the actual reasoning and scheduling components of this predicate have been completed. Once the list of goals has been unified, the reduction of the goal-plan trees can start by removing the branch options as described above. The predicates for the three types of reasoning as shown above can be added or removed as necessary, depending on the types of reasoning desired.

Once all the plans that are not required have been removed, either because of branch options, positive interactions or limited resources restricting the number of goals that can be adopted, the finite domain variables for each of the remaining plans are asserted as global variables. This is contained within a `varSetup` predicate that iterates through the list of all the remaining plans asserting the global variables with the domain ranging from 0 to the number of plans now

being considered, i.e. the length of the list of plans. After this has been successfully completed, it is then possible to start applying the constraints that restrict the assignment of the values from the domains to the variables. This starts with the scheduling based on the tree structure and finishes with the negative interference reasoning (Section 2.5), when this is incorporated into the types of reasoning being performed. At this point, the labelling of variables with values is to be performed, so the `varResult` predicate simply collects all of the finite domain variables back into a list which is then passed on to the finite domain labelling predicate (`fd_labeling`). This predicate is part of the prolog library for solving finite domain constraint satisfaction problems, and provides a selection of heuristics for ordering the variables; the heuristic selection is given as a parameter to the predicate along with the list of variables.

While this design achieves the objectives of representing and reasoning about the goal-plan tree, it may still be possible to optimise some of the constraints in order to improve their efficiency, thereby reducing the length of time taken for a solution to be found.

2.3 Consumable-Resource Reasoning

The reasoning described here is limited to that of consumable resources rather than reusable resources. The purpose of the reasoning is to restrict the number of goals adopted to those that can be achieved with the amount of consumable resources available and to endeavour to make the best use of those resources through the careful selection of plans when there is a choice between which plans to use in order to achieve the desired result. The reasoning about consumable resources makes use of a small amount of generated summary information to perform this reasoning.

As described in the section above, the resource requirements for each plan are represented by a list of pairs consisting of resource type and quantity required. The total available resources for each type are each defined using a `resource` predicate. This predicate is defined to be dynamic so that when reasoning about resources the quantity available can be updated with the new quantities as they are consumed.

The first part of the resource reasoning is incorporated into the constraint reasoning for the selection between lists of plan options for achieving a goal or subgoal. For each of the plans listed as being an option, a summary of the resource requirements for the sub-tree with the plan at its root is generated. At this point, a single number for all the resource quantities required regardless of resource type is used to decide which plan to use. It is possible to extend the reasoning here to incorporate weightings into the summation of resource requirements in order to indicate preference for the use of certain resources over others.

When this type of reasoning is included, the definition of the `branchStrip` predicate shown above is extended to refer to a predicate that pairs the summary resource requirement with each plan in the list of options. The list of plan options is sorted so that the subgoal branches nearest the leaves at the bottom of the tree

are considered first. This is to reduce the number of plans being considered at each iteration through the list and to allow for simpler predicates summing the resource requirements as they do not need to consider branches at lower subgoals. Once the list of plan options paired with resource requirements is formed, it is then sorted into order of increasing resource requirements so the first element in the list is the preferred plan and the remaining plans can again be retracted.

```
branchStrip([]).
branchStrip([H|T]):-
    branchList(H,L),
    sort(L,[_|T2]),
    rmBranch(T2),
    branchStrip(T).

branchList([],T):-T=[].
branchList([P|T],T1):-
    branchList(T,T2),
    subtree(P,X),
    resAll(S,X),
    append([S/P],T2,T1).
```

The `resAll` predicate starts by producing a single long list of the resource requirements for each plan. For each plan, this takes the pairs representing the type of resource and quantity required and appends them to a list of all the resource requirements for the sub-tree being considered. Once all the resource requirements have been compiled into one list, this is sent to a summing predicate to simply add together all the quantities to produce a total resource requirement. It is in this final predicate where weightings could be included, if necessary, to indicate any preferences for which types of resources should be saved or used the most.

```
resAll(S,Ps):-
    resourceList(L,Ps),
    resSum(S,L),!.

resourceList(L,[]):-L=[],!.
resourceList(L,[SG|T]):-     % Only interested in plans
    node(SG,_),
    resourceList(L,T).
resourceList(L,[P|T]):-
    node(P,_,_,_,R),
    resourceList(L1,T),
    append(L1,R,L).

resSum(S,[]):- S=0,!.
resSum(S,[_/X|T]):-
    S#=X+S1,
    resSum(S1,T).
```

After the plan options have been removed, the resource reasoning is next used to consider which goals can be safely adopted given the quantity of each resource available. The reasoning is performed in this order firstly to reduce the number of plans being considered and secondly to allow the summary information generated for reasoning about goal adoption to represent the actual requirements of the goal.

The list of top-level goals can be sorted in the same manner as the list of plan options for selecting the plans or, in this case, goals with the lowest resource requirements. To do this, the first step, as before, is to generate the list of plans in the tree for each goal. This can be performed using the `branchList` predicate with a list of the top-level goals. This will pair up each of the goals with a number representing the sum of resource requirements regardless of type. It is possible to apply different orderings to the list of goals to indicate the importance of a goal, thereby preferring to complete less goals of greater importance than to achieve more goals of less importance. If the order in which the goals are considered for adopting is not important, or if the order is predefined as the order in which the goals were defined, this step can be skipped. This will also provide a decrease in the number of steps and hence the length of time taken to evaluate the problem each time a solution is to be found. In the evaluation of this approach, both sorting and ordering were included in the reasoning.

The main reasoning about resources for goal adoption requires summary information broken down by the different types of resources required. This is so that the reasoning can check that there is actually sufficient resources available for each goal to be adopted. For each goal in the list, the summary information separating the different types of resource information is generated. While the `resAll` predicate produces a combined summary of each of the resource types into one number, the `resType` predicate used here keeps the different types of resources separate when generating the summary information. The summary information produced by the predicate `resType` is an unsorted list containing each of the resource types and the quantity of it required by the goal, for example `S = [r3 / 6, r2 / 5, r1 / 7, r5 / 0, r4 / 0]`. From this list, each of the types of resource is extracted and compared to the available quantity of that resource.

```
resReason(G):-
   goalPlans(G,P),
   resType(S,P),                % generate resource summary by type
   member(r1/A,S),              % unify the resource values
   member(r2/B,S),
   ...
   resource(r1,RA), RA#>=A,     % check sufficiently available
   resource(r2,RB), RB#>=B,
   ...                          % reserve resources
   retract(resource(r1,RA)),
   NewRA #= RA-A, asserta(resource(r1,NewRA)),
   retract(resource(r2,RB)),
```

```
NewRB #= RB-B, asserta(resource(r2,NewRB)),
...
```

If each type of resource has sufficient resources available then the predicate `resReason` will succeed and the quantity of each of the resources available will be lowered accordingly. If one or more types of resource has as insufficient quantity available then the predicate will fail and the *if-then-else* construct from which the predicate was queried (`resReason(G) -> true; strip(G)`) will step to the *else* component where the whole goal will be dropped in the same way as for removing the sub-tree of a plan that is not required. After all the goals have been considered, adopting those that are safe to start, and removing those which are not, the reasoning then returns to the core part of the goal-plan tree representation to schedule the plans for the goals that have been adopted.

2.4 Positive Interaction Reasoning

The positive interaction reasoning attempts to identify plans in different goal-plan trees that can be "merged", as they produce the same effects. When referring to plan merging, it is actually possible to achieve the effects by only using one of the two plans. By doing this, the number of plans required to achieve all the goals adopted can be significantly reduced, especially as the sub-trees of the plans that are not used are also removed when the two plans are merged. If the interaction between the goals occurs at high levels of the goal-plan trees, i.e. near the root with each plan itself having a large sub-tree, then the impact of the merging is particularly significant. Figure 3 illustrates where two plans will achieve the same effect, so only one of the plans is need to reach the desired state.

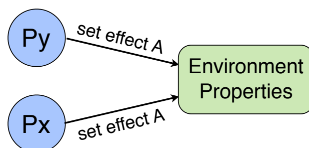


Fig. 3. Illustration of positive interaction

To perform the reasoning in Prolog, a predicate is defined that identifies pairs of plans that produce the same effects by checking that the lists of effects for the two plans are equivalent. This starts by unifying two plans and the list of effects generated by each of the plans, checking that the two plans are not the same plan. The reasoning cycle in Prolog when requested for all pairs of positively interacting plans will iteratively test every pair of plans. For pairs of different plans, the effects of the plans are considered to identify if there is any possibility of merging them. Firstly, it is checked that the list of effects for the first plan is not empty, otherwise all plans that themselves do not achieve effects could

be included for merging. Where an effect is produced by `Px`, the list of effects for the two plans are compared to see if they are equivalent. If so, then with all the constraints satisfied, the pair of plans is returned as a pair of positively interacting plans that can be merged. If the effects are not equivalent, then the solver backtracks to try another pairing until all possible pairings have been tested.

```
pos(Px,Py):-
    node(Px,_,_,XEffects,_),
    not(XEffects=[]),
    node(Py,_,_,YEffects,_),
    Px\=Py,
    seteq(XEffects,YEffects).
```

The `findall([Px,Py], pos(Px,Py), Merge)` predicate is used to generate a list all the pairs of plans where it is possible for them to be merged. The template used to form the list from the solutions to the `pos(Px,Py)` predicate creates a sublist for each solution pair of plans. The complete list of positively interacting plans is then used to select and remove plans that are not needed as the effects they produce are duplicated by other plans. By default, the second plan in the pair of interacting plans is retracted, however this is not always the case.

While in the positive interaction reasoning considered here all the effects in the list must match for the plans to be considered for merging, it is also possible to consider a weaker version of positive interaction where only some of the effects match. In this case, in order to ensure that a plan that is kept from the merging with another plan is not then deleted by a later merging, the plan is "marked". This is done by asserting the predicate `mark(Plan)` for each of the plans that have been kept from a merged pair. When a pair is first considered, it is checked to see if either plan is already marked. If both plans are already marked, then neither plan can be safely removed as it is possible that the intersecting effect that was used to identify the two plans as positively interacting is different to the intersecting effects from the interactions where they have already been "merged".

As the reasoning here checks that the effects are equivalent, it is not necessary to check if one or both plans are already marked. This is because if one plan is marked, and has appeared in more than one positive interaction then the effects of three or more plans must all be equivalent, therefore only one plan is still needed to achieve the effects on behalf of all of the plans. However, as merges could have occurred within the sub-tree of one or both of the interacting plans, it is still necessary to mark the plan kept from a merge to ensure it does not get removed as part of a sub-tree.

The `posScheduler` predicate defined below starts by checking that the two plans both still exist, i.e. that one or both have not already been removed by other merges. The sub-trees of each plan are then generated to check for any marked plans within the sub-trees that could prevent one of the plans from being removed in a merge. If just one of the plan's sub-trees contains a marked plan,

then that plan can be kept while the other is retracted, otherwise neither plan and their sub-trees can be removed.

```
posScheduler([]).
posScheduler([[P1,P2] | T ]):-
   node(P1,_,_,_,_), node(P2,_,_,_,_),
   subtree(P1,X), subtree(P2,Y),
   not((member(XP,X), mark(XP));
       (member(YP,Y), mark(YP))),
   ((not(member(XP,X), mark(XP)), asserta(mark(P1)), strip(P2));
   (not(member(YP,Y), mark(YP)), asserta(mark(P2)), strip(P1))),
   posScheduler(T).
```

When the reasoning about positive interactions is combined with that of reasoning about consumable resources, then the selection for which plan to keep and which plan to drop is influenced by the summary resource requirements for the sub-tree of each plan. In this case the predicate `resAll` is used to produce the summary information for the sub-tree of each of the two plans. The plan with the lower resource requirements is then kept when there is a free choice between the two plans as neither sub-tree contains any marked plans.

The positive interaction reasoning is incorporated into the set of constraints after the branch options have been removed. This is to reduce the number of matches as the branches provide different sets of plans for achieving the same effects within a goal-plan tree.

2.5 Negative Interference Reasoning

While the reasoning about positive interaction identifies plans that produce the same effects, the reasoning about negative interference identifies sets of three plans where one plan generates the effect required by the second plan, and the third plan produces an opposite effect that if it were executed between the first two would cause interference. This can be thought of as a causal link between the first two plans, which the third plan would break. Figure 4 illustrates a case of negative interference.

In Prolog, in order to identify the negative interactions between plans, the `neg(Px,Py,Pz)` predicate is defined to find pairs of plans that have causal links and the plans that can interfere with those links. `Px` is the plan that starts the causal link by producing the desired effect required as a precondition for plan `Py`. Once `Py` has executed, it is assumed that the effect is no longer required, so can

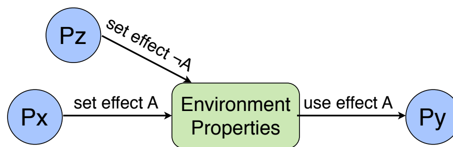


Fig. 4. Illustration of negative interference

be safely altered by other plans such as Pz. If however Pz attempts to execute between Px and Py, then this will cause interference, possibly leading to plan and then goal failure. As with the positive interaction reasoning, it is important to check that the plans are all different before comparing the preconditions and effects of the plans. To compare the effects, it is important to split up the pair notation for representing the effects of plans into the two component parts, the factor identifier and the value representing its current state (e.g. e1/7). The member(Element, List) predicate, in the reasoning predicate shown below, unifies properties of the environment that are common to all three plans but where the value assigned to that property is different in the interfering plan to the value used by the linked plans.

```
neg(Px,Py,Pz):-
    node(Px,_,_,XEffects,_),
    node(Py,_,YPrecon,_,_), Px\=Py,
    node(Pz,_,_,ZEffects,_),
    Px\=Pz,Py\=Pz,
    member(V/N1,YPrecon),
    member(V/N1,XEffects),
    member(V/N2,ZEffects),
    N1#\=N2.
```

This predicate is again queried with the findall([Px,Py,Pz], neg(Px,Py,Pz),Neg) predicate to generate a list of all the possible instances of the interference so they can be scheduled to ensure the interference is avoided. For this, the interfering plan either needs to be scheduled to execute before the other plans or after both have executed so the effect is no longer required. This is handled by the negScheduler predicate shown below.

```
negScheduler([]).
negScheduler([[Px,Py,Pz]|T]):-
    g_read(Px,A),
    g_read(Py,B),
    g_read(Pz,C),
    A#<#B,(C#<#A;C#>#B),
    g_assign(Px,A),
    g_assign(Py,B),
    g_assign(Pz,C),
    negScheduler(T).
```

The negScheduler predicate refers to the finite domain global variables that have been defined for representing the domain of values that can be assigned to each of the variables representing the plans for generating a schedule. The plan producing the effect (Px) must always occur before the plan using the effect (Py). However, it is possible to schedule the interfering plan (Pz) to either execute before Px or after Py, as long as it does not execute between the two plans.

The reasoning about negative interference is incorporated into the set of constraints after the tree scheduling has been performed. This is to ensure the

minimum number of plans are considered as the evaluation of the `neg(Px,Py,Pz)` predicate considers all the possible combinations of three plans. In addition, the main purpose of the negative reasoning is to schedule potentially interfering plans to ensure they do not interfere, rather than reducing the number of plans, so this "scheduling" is performed after all the surplus plans have been removed and the schedule refined based on the constraints in the tree structure.

3 Experimental Results

To compare the performance of the three types of reasoning under different conditions, three different tree structures were used; a deep tree, a broad tree and a tree that is part way between the two (referred to as the general tree structure). The results presented here are a subset of a large set of experiments comparing a wide range of variables covering goal-plan tree size, goal interaction levels and resource availability amongst others. The aim of the experiments was to stress test the approach described here and compare it to the approach described in [5,6], to identify settings where one approach was able to perform better than the other. Each of the types of reasoning was considered independently before combining all three together. The performance of the two reasoning approaches was also compared to the performance without any reasoning, simulated by a Petri net model with the reasoning removed. An example of a more concrete application to which this reasoning could be applied is presented in [5], where a simplified Mars Rover is modelled. While the approach here can be applied to this example, the results presented here are aimed at illustrating performance under highly constrained conditions with a large number of substantially sized goals.

In order to fully evaluate the performance of this approach and compare it to other approaches, a set of large goal-plan trees has been designed with high levels of interactions between them and heavy resource requirements. The goals were designed to test different properties of the reasoning, for example there is a deep tree structure that has very little branch options, and was designed to test the effect of depth and size of sub-trees on the reasoning. Conversely, a broad tree structure containing a lot of branch options has been designed to test the ability of the two approaches to handle branches and select the best options where appropriate. A third tree structure was also used to test the scalability of the two models, so it contained nearly 100 plans and was used in experiments focused on increasing the number of goals.

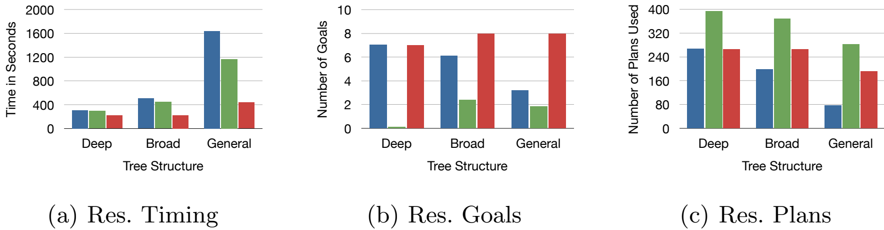
An overview of the results are presented below, summarising results across the different tree structures. The graphs below combine the results for common settings in each of the tree structures for each type of reasoning, individually and combined. They show the results for experiments using a medium-sized deep and broad tree (∼50 plans) or a large tree (∼100 plans) from the general tree structure, 20 goals, low level resource availability, positive interaction at a high level in the goal-plan tree, negative interference over a long duration and high goal interaction. When showing the timings, the load timings for both models

Fig. 5. Legend for graphs comparing performance over the three different tree structures

are included in the graphs, as well as the run times for the two approaches, as the run time for the Petri net model was very short, but the load time was quite long. The legend for the graphs is shown in Figure 5.

3.1 Reasoning about Consumable Resources

While the Petri net model was able to match the number of goals achieved by the constraint model in the deep tree, the performance in the broad and general trees was much worse, see Figure 6. In comparison, the random Petri net model was able to achieve more goals in the broad and general trees than in the deep tree. The timings for the Petri net model were greater than those for the constraint model when including loading times, especially in the large-sized general tree structure experiments. Overall, the constraint model gave better results both in terms of time and number of goals achieved when there is limited availability of consumable resources, especially in trees where there is a large amount of branching.

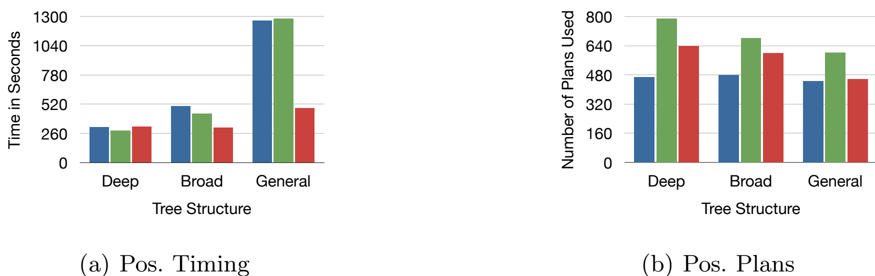


(a) Res. Timing (b) Res. Goals (c) Res. Plans

Fig. 6. Comparison results for reasoning about resources across the three tree structures

3.2 Reasoning about Positive Interaction

When reasoning about positive interaction, the Petri net was able to generate better results based on the reduction in the number of plans used in each of the tree structures, see Figure 7. Comparing the timings here shows that while the time taken between the Petri net and the constraint models was the same for the deep tree, the Petri net model took longer to load in the experiments for the other two tree structures, especially the large tree size of the general tree structure. When the number of plans used is the key criteria, then the Petri net



(a) Pos. Timing

(b) Pos. Plans

Fig. 7. Comparison results for reasoning about positive interaction across the three tree structures

model performs better; however, if time is critical then the constraint model can produce results slightly faster when reasoning about positive interaction is applied.

3.3 Reasoning about Negative Interference

While the reasoning about negative interference was the most time consuming of all the three types of reasoning, it is perhaps the most critical when comparing the results achieved to those produced when no reasoning is included, as illustrated in Figure 8. In this case, the time taken by the Petri net even when the load times are included is much shorter for the experiments on the deep and broad tree structures. However, the loading time on the large-sized tree for the general tree structure does take longer than the constraint model in this setting. Overall, the Petri net model offers better results here, especially with the small and medium tree structures.

3.4 Combined Reasoning

When combining the three types of reasoning together, the number of goals achieved increased, especially in the deep tree where a large number of plans were saved by the positive interaction reasoning, as shown in Figure 9. The resources that would have been consumed by these plans were then available for use in achieving other goals. This combined effect is less noticeable in the broad and general trees. However, the constraint model was generally able to make the most optimisations here. The exception to this is that as the availability of the resources was increased in the general tree structure, the number of goals started and hence the plans interacting increased, resulting in more plans not being used so more resources being saved for use in achieving further goals. In the high level resource availability for the general tree, this lead to all goals being achieved by the Petri net model.

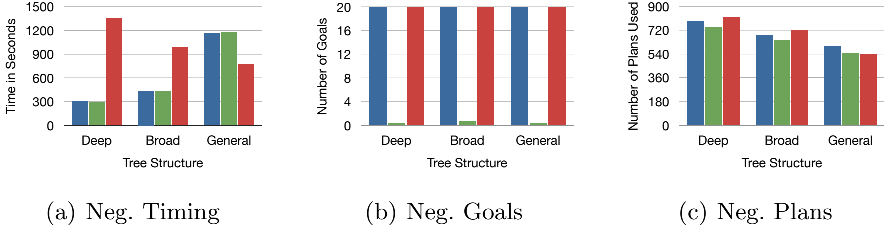


Fig. 8. Comparison results for reasoning about negative interference across the three tree structures

In the experiments for the deep tree, the Petri net timings even when including the loading times were quite similar to those for the constraint model, however in the experiments for the other two tree structures, especially the large-sized general tree, the time taken for loading the Petri net model was greater than the time taken for the constraint model to find a solution. Despite the additional time taken for the reasoning in both models, the benefits gained from performing the reasoning over those shown in the random Petri net model indicate that it is worth considering taking the time to find a good solution. In highly dynamic environments, there may not be the time available to consider this as too much would have changed by the time a simulation had finished.

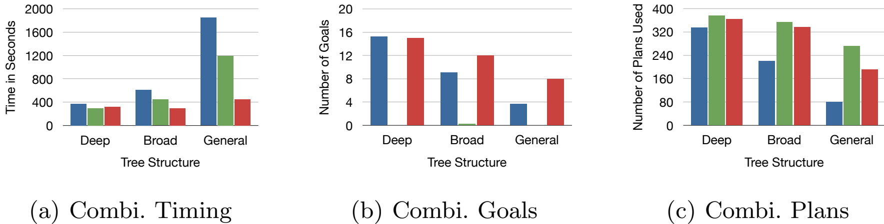


Fig. 9. Comparison results for combined reasoning across the three tree structures

When increasing the number of goals used in the general tree structure from 20 to 50, the Petri net model became too large for the Petri net editor (Renew [4]) to load the model, while the constraint-model was able to continue to reason about up to 75 goals. However, it took 9hrs to find a solution, so further increases in the number of goals were not tested. It should be noted that in current agent applications, the size and number of goals tends to be significantly less than those tested here. This means that the time taken for the reasoning can be within acceptable ranges for practical purposes, at least in some applications.

4 Conclusions and Future Work

In this paper we have presented a specific Prolog implementation (with constraint solving) that could be used to solve the goal-plan tree problem, which has then been compared to a second specific implementation based on Petri nets. While these two approaches are both capable of solving the same problems, the techniques they use are different and as a result the solutions they offer can vary. For example, the approach here controls the sequence in which goals and plans are considered. If a set of goals is evaluated again, it will be evaluated in the same order and give the same answer each time. However, in the Petri net approach, no exact order in which goals and plans are evaluated is set, so in each evaluation the order can vary. This can lead to differences in the results returned, particularly when resources are constrained. It is possible that variations in the used of the underlying techniques would offer different advantages, however further experimental comparisons such as the one performed here would be needed.

The approach presented here has been experimentally compared to the Petri net approach described in [6]. The complete results from the comparison can be found in [7]. The aim of these experiments was to test the two approaches under highly constrained conditions and to identify situations where one approach may be better suited over the other. The differences between these two approaches can be beneficial in different situations and conditions where some properties of one or the other approach may be preferable.

The results presented here show that while the Petri net model has faster running times, it also has the slowest loading times with the greatest memory usage once loaded. One of the side effects of this is that, as the size of the goal-plan trees or the number of top-level goals increases, the load times rapidly increase until the application running the Petri net simulations is no longer able to load the Petri net goal-plan tree representation. Refinements and changes in the way the goals are represented may reduce the problem allowing greater numbers of goals to be handled in the Petri net model. Similarly, it is possible that refinements in the efficiency of the Prolog constraints used in the constraint-based model may improve the performance of this model as well.

In some cases the Petri net model can give better results over the constraint-based model. In particular, when reasoning about positive interactions between the goals, the Petri net model gives better results in terms of the number of plans used, and when reasoning about negative interference, the Petri net model also gives faster results for successfully achieving all goals, even when including the loading times. The structure of the tree also affected the performance of the two reasoning models, with the constraint-based model performing better when applied to reasoning about the limited availability of resources in trees where there is a large amount of branching and little depth.

Where the ability to reason about large numbers of goals is required, especially for large sized trees, the constraint model demonstrated that it was able to scale and find solutions to larger problems. However, the trade-off comes at the time taken, for example taking 9 hours to reason about 75 goals in one setting.

The approach described here has been compared based on the individual types of reasoning and the combined reasoning. While in most cases it makes sense to combine all three types of reasoning, there may be application areas where only one is needed. For example, in applications where there is limited availability of consumable resources but very little interaction between the goals it may only make sense to use the resource reasoning. Similarly, in applications where there are a lot of common goals to achieve the same effects, and abundant resources it may be better to use only the positive interaction reasoning. In applications where there is likely to be a lot of conflict between the goals or where it is more critical that all the goals are achieved, but again with abundant resources, it may be sufficient to just apply the negative interference reasoning.

In conclusion, the following recommendations can be made to agents about which model they may wish to consider depending on their specific circumstances:

– When just considering resource reasoning, if the goal-plan trees contain a lot of branching then the constraint-based model gives better results in terms of goals achieved.
– When just considering positive interaction reasoning, the Petri net model gives better results for all goal-plan tree structures in terms of the reduction in plans used.
– When just considering negative interaction reasoning, the Petri net model gives better results for all goal-plan tree structures in terms of the time taken to perform the reasoning.
– When considering the combination of all three types of reasoning, the constraint-based model gives better results in terms of goals achieved except when there is high resource availability, in which case the Petri net model performs better.
– When there are a large number of large goals (i.e. 50 or more goals containing more than 100 plans), only the constraint-based approach is able to perform the reasoning, although it will take considerable time to find a solution.

The reasoning about resources that has been considered here has focused on consumable resources that are limited in their availability. Another type of resource that is often used are reusable resources, such as communication channels. A model was shown in [6] of how this could be incorporated into the Petri net model, and constraints could be added into the constraint-based approach to prevent two plans attempting to use the same reusable resource at the same time. This was not initially included as the use of these resources can be easily scheduled, while the use of consumable resources has greater restrictions applied to it. The reasoning about consumable resources is also the more difficult of the two types of resources to implement, being possible to later incorporate the reasoning for reusable resources easily. In addition, when considering consumable resources, all the goals are currently assumed to consume resources without any goals to recharge them or to create more resource instances. The Petri net approach and to some extent the constraint-based approach are however robust

enough to handle this, at least in a simplistic manner. However, further work to extend both approaches to allow for more generic maintenance goals rather than only achievement goals is required.

References

1. Clement, B.J., Durfee, E.H.: Identifying and resolving conflicts among agents with hierarchical plans. In: Proceedings of AAAI Workshop on Negotiation: Settling Conflicts and Identifying Opportunities, Technical Report WS-99-12, pp. 6–11. AAAI Press, Menlo Park (1999)
2. Clement, B.J., Durfee, E.H.: Theory for coordinating concurrent hierarchical planning agents using summary information. In: AAAI 1999/IAAI 1999: Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence, pp. 495–502. American Association for Artificial Intelligence, Menlo Park (1999)
3. Clement, B.J., Durfee, E.H.: Performance of coordinating concurrent hierarchical planning agents using summary information. In: Proceedings of 4th International Conference on Multi-Agent Systems (ICMAS), pp. 373–374. IEEE Computer Society, Boston (2000)
4. Kummer, O., Wienberg, F., Duvigneau, M.: Renew – the Reference Net Workshop, Release 2.1 (May 2006)
5. Shaw, P.H., Bordini, R.H.: Towards alternative approaches to reasoning about goals. In: Baldoni, M., Son, T.C., van Riemsdijk, M.B., Winikoff, M. (eds.) DALT 2007. LNCS (LNAI), vol. 4897, pp. 104–121. Springer, Heidelberg (2008)
6. Shaw, P., Farwer, B., Bordini, R.H.: Theoretical and experimental results on the goal-plan tree problem. In: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, vol. 3, pp. 1379–1382 (2008)
7. Shaw, P.H.: Reasoning about Goal-Plan Trees in Autonomous Agents: Development of Petri net and Constraint-Based Approaches with Resulting Performance Comparisons. PhD thesis, School of Engineering and Computing Sciences, University of Durham, UK (January 2010)
8. Thangarajah, J., Padgham, L., Winikoff, M.: Detecting and avoiding interference between goals in intelligent agents. In: Proceedings of 18th International Joint Conference on Artificial Intelligence (IJCAI), pp. 721–726. Morgan Kaufmann, Acapulco (2003)
9. Thangarajah, J., Padgham, L., Winikoff, M.: Detecting and exploiting positive goal interaction in intelligent agents. In: Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 401–408. ACM Press, New York (2003)
10. Thangarajah, J., Winikoff, M., Padgham, L.: Avoiding resource conflicts in intelligent agents. In: Proceedings of 15th European Conference on Artifical Intelligence (ECAI 2002), IOS Press, Amsterdam (2002)