RI 06002 15 February 2006 Computer Science

IBM Research Report

Applying Planning in Composition of Web Services with a User-Driven Contingent Planner

Anupam Mediratta and Biplav Srivastava
IBM Research Division
IBM India Research Lab
Block I, I.I.T. Campus, Hauz Khas
New Delhi - 110016. India.

Applying Planning in Composition of Web Services with a User-Driven Contingent Planner

Anupam Mediratta and Biplav Srivastava
IBM India Research Laboratory
Block 1, IIT Campus, Hauz Khas
New Delhi 110016, India
{anupamme, sbiplav}@in.ibm.com

Abstract

Planning, the discipline of AI which focuses on synthesis of action sequences (or plans*) to satisfy goals, has an important role to play to make Web Services Composition a practical reality. While the scope of web services composition spans from creation of business process functionalities, to developing executable workflows that capture non-functional (e.g. QoS) requirements, to deploying them on a runtime infrastructure, we are interested in using planning to generate the initial plan templates that satisfy the functional goals. These plans can be made optimal and deployable using deployment and runtime requirements known later. Planning needs to be efficient in this interactive and uncertain domain, the domain could be incompletely modeled, the user has hard and soft constraints, and the number of web services is large. We show how a contingent planner can be adapted to this end by allowing for user inputs that are then employed to efficiently finds plans that matter most to them. Moreover, the planner is useful for interactive planning applications in general.*

1 Introduction

Web services have received much interest in industry due to their potential in facilitating seamless business-to-business or enterprise application integration. A web service composition and execution system (WSCE or end-to-end web services composition) can help automate the process, from creating business process functionalities, to developing executable workflows that capture non-functional (e.g. QoS) requirements, to deploying them on a runtime infrastructure. Initial approaches for web services composition viewed it as a planning problem where the operations of the available web services are actions, and the goal state is the specification of the composite service[14, 8]. A plan for the planning problem would realize the composite service assuming all other considerations for realizing the service were benign. However, the problem of WSCE cannot be seen as a one-shot plan synthesis problem defined with explicit goals but rather as a continual process of manipulating complex workflows, which requires solving synthesis, execution, optimization, and maintenance problems as goals get incrementally refined [16].

Recently, we have proposed the *Synthy* approach for end-to-end composition of web services with the aim of automating service/application creation[1]. It is two-step methodology that differentiates between web service *types* and *instances* and decouples web service composition into logical and physical composition stages that address complementary integration issues. The first stage focuses on the feasibility of functional composition using generative planning techniques while the second stage deals with efficient execution of the resulting composition using optimization techniques. A prototype is also built that demonstrates this methodology in a domain-specific scenario.

In this paper, we identify the unique characteristics of generative planning in end-to-end composition of web services and propose a planner with novel techniques for solving it. These characterstics are: (a) *The nature of planning is contingency planning (CP)*. The value of all logical terms may not be known in the initial state but they can be found at the runtime using sensing actions. (b) **Domain may be incompletely modeled so all branches may not lead to goal**. While a conventional sound planner will not give a plan in such a domain since it ensures that all branches can lead to goal, the user would benefit from incomplete plans which act as suggestions to enhance the domain model. (c) *The user may not be interested in all branches* - which are exponential in the number of unknown terms - but only in specific branches. For the rest, he may insert a default branch. The nature of planning is thus interactive and limited contingency planning. (d) *There can be soft constraints* which the user of the end application would prefer to be followed. (e) *The number of web services instances can be large* (i.e.,in the range of 10000's) and the planner should be scalable.

To meet the above challenges, we have devised a novel user-driven search-control methodology for CP which takes input from user and then uses them to efficiently focus the search. These inputs are: conditions of interest, the type of plan desired and the number of conditions to handle. Our approach employs user-inputs on the AND-OR graph of CP's belief states to prune space in the AND part of the graph and additionally uses the well known planning-graph (PG) based heuristics to do the same in the OR part. The approach efficiently finds contingent plans focusing on user interest and it is complementary to recent utility based methods for contingency selection. We have implemented such a contingent planner in the Planner4J[15] planning framework and will refer to it by P4J-CP.

For limited CP, there has been some work done in selecting a subset of specific contingencies for plan generation. In [11], a planner is proposed which identifies the contingencies that contribute most to the plan's success and iteratively makes re-planning strategies for failure of such contingencies up to some limit. In [9], the utility of the contingencies is modeled and the planner tries to optimize the overall utility while addressing the subset of contingencies. Our motivation is that in interactive applications, the user can play a role in driving the search in contingent planning so that a timely result is obtained for the conditions the user is interested in.

Approaches that take user input to control the search space can be seen in two categories. In the first category, user input specifies the preference for the sequence of actions under some condition which guides the planner to use particular sequences of actions if corresponding condition becomes true[5]. In contrast, in the second category, user input specifies the conditions which define good and bad states so that during search, planner can use this information to focus the search on good states[2]. Our approach falls in the second category. While there are more expressive approaches than ours to seek user inputs [6, 12], our focus in on the orthogonal and unexplored area of how to use input specifications when domain desciptions may be incomplete and plans can have both sound and unsound segments that users may want.

In the next section, we formalize the contingent planning problem and give details of our solution approach. We then demonstrate how the output contingent plan varies with user specification inputs. Then, we present evaluation on the effectiveness of our approach and follow it with conclusion and future work.

2 Formalizing the Contingent Planning Problem

Here, we formalize the CP problem under discussion, the type of plans that P4J-CP generates and illustrate with an example.

Definition 1 *A contingent planning problem P is a 4-tuple (BS_I, BS_G, A, L) where BS_I is the initial belief state, BS_G is the partial description of the goal belief state, A is the set of actions and L is the set of binary valued atoms.*

To model uncertainty, we further classify L into:

- L_O, the set of observable atoms. The truth value of $l_i \in L_O$ may not be known in BS_I.
- L_S, the set of supporting atoms. The supporting atoms are introduced to handle uncertainty about $l_i \in L_O$. For each member of L_O, o_i, two supporting atoms, *unknown-o_i* and *willknow-o_i* are in L_S. No other atom is a supporting atom and hence $|L_S|$ = 2 $*$ $|L_O|$. The value of $l_i \in L_S$ is known in BS_I.
- L_R, the set of regular atoms. All other atoms (L - (L_O $\cup$ L_S)) belong to this set and the value of these atoms is known in BS_I.

For any observable atom o_i, the following axioms govern its values and that of the supporting atom:

- $o_i =$ TRUE $\vee$ $o_i =$ FALSE $\rightarrow$ *unknown-o_i* $=$ FALSE
- *willknow-o_i* $=$ TRUE $\rightarrow$ $o_i =$ TRUE $\vee$ $o_i =$ FALSE
- *unknown-o_i* $=$ TRUE $\rightarrow$ *willknow-o_i* $=$ FALSE and *unknown-o_i* $=$ FALSE $\rightarrow$ *willknow-o_i* $=$ TRUE.

Definition 2 *A Belief State BS is a set of literals, where a literal is either an atom or its negation. Hence, $BS = \{l_i \mid l_i \vee \neg l_i \in L\}$*

Definition 3 *A World State WS is belief state with no* unknown *supporting atoms. Hence, for all $o_i \in L_O$, no literal in WS is of the form* unknown-o_i, *i.e.*, $WS = \{l_i \mid \forall o_i \in L_O, l_i \neq$unknown-$o_i \vee \neg l_i \neq$unknown-$o_i\}$

For any observable atom o_i, if *unknown-o_i* is TRUE, then the value of o_i is unknown. Since each o_i is binary valued, a BS with $N unknown - o_i's$ encodes 2^N world states. Since we do not encode each of the WS explicitly, our representation is compact like planner PKS [12] and much smaller than other approaches which encode them explicitly e.g. planner MBP[4].

Each action in A is either a non-sensing action or a sensing action.

Definition 4 *A non-sensing action A_i is specified by a three tuple: (A_i^{Pre}, A_i^{Add}, A_i^{Del}), where each element of A_i^{Pre}, A_i^{Add} and $A_i^{Del} \in L_R \cup L_O$.*

Definition 5 *A sensing action corresponding to an observational atom o_i,* Sense-o_i, *is specified by:*

*Action (*Sense-o_i*)*
:Precondition: $\{unknown - o_i\} \cup P_i^{Pre}$, *where* $P_i^{Pre} \subseteq L_R \cup L_O$
:Add Effect: willknow-o_i
:Del Effect: unknown-o_i

In the definition of sensing action, P_i^{Pre} is the list of other literals which might occur as precondition besides the necessary precondition: *unknown-o_i*. The application of *Sense-o_i* results in the runtime value of o_i to be known. Since it is binary valued, the plan branches to two states: one contains the o_i as TRUE and other as FALSE. Its application does not lead to change in the value of any atom belonging to the set L_R, i.e., the world is unchanged.

As an example, let there be three stores: A, B and C. Store A sells flower F1, store B sells flower F2 and store C sells both flowers F1 and F2. We have sensing actions for each pair of store and flower it sells and they sense whether the flower is in stock at the store or not. Then, there are packing, billing and dispatch actions to fulfill the order. The action specifications are shown in Table 1. The planner, like all AI planners, uses preconditions and effects. A WSDL-described web service has only inputs and outputs, while an OWL-S described (semantic) web service has inputs, outputs, preconditions and effects. We use the latter format for exposition but note that WSDL-described specifications can be translated to preconditions and effects[1]. In a problem (referred by *ExampleProblem* later), suppose the goal is to dispatch two kinds of flowers: F1 and F2. Hence, BS_I = { *unknown-A-F1, unknown-B-F2, unknown-C-F1, unknown-C-F2*} and BS_G = { *dispatch_F1, dispatch_F2*}.

We now define different kind of plans.

Definition 6 *Potential Plan: Any sequence of actions is a potential plan.*

Definition 7 *Potential Contingent Plan: These are the rules describing a potenital contingent plan:*

- *A Potential Plan is a potential contingent plan.*
- *If an action a is a sensing action, and w_1,...., w_n are conjunctions of observable atoms and c_1,...., c_n are corresponding potential contingent plans, then so is "a;case ($w_1 \rightarrow c_1$, ..., $w_n \rightarrow c_n$)".*

[1]The representation of actions is the same as that of MBP. It can be naturally transformed to other planner representations.

Name: Sense-A-F1 *Input*: FlowerName F1 ; *Output*: *Precon*: unknown-A-F1 ; *Effect*: willknow-A-F1	*Name*: Sense-B-F2 *Input*: FlowerName F2 ; *Output*: *Precon*: unknown-B-F2 ; *Effect*: willknow-B-F2
Name: Sense-C-F1 *Input*: FlowerName F1 ; *Output*: *Precon*: unknown-C-F1 ; *Effect*: willknow-C-F1	*Name*: Sense-C-F2 *Input*: FlowerName F2 ; *Output*: *Precon*: unknown-C-F2 ; *Effect*: willknow-C-F2
Name: Pack-C (**S1**) *Input*: C-F1, C-F2 ; *Output*: Packed_C_F1, Packed_C_F2 *Precon*: C-F1 $\vee$ C-F2 ; *Effect*: C-F1 $\longrightarrow$ Packed_C_F1, C-F2 $\longrightarrow$ Packed_C_F2	*Name*: Pack-A (**S2**) *Input*: A-F1 ; *Output*: Packed_A_F1 *Precon*: A-F1 ; *Effect*: A-F1 $\longrightarrow$ Packed_A_F1
Name: Pack-B (**S3**) *Input*: B-F2 ; *Output*: Packed_B_F2 *Precon*: B-F2 ; *Effect*: B-F2 $\longrightarrow$ Packed_B_F2	*Name*: Bill-Card-C (**S4**) *Input*: Packed_C_F1, Packed_C_F2 ; *Output*: billed-Card_F1, billed-Card_F2 *Precon*: Packed_C_F1 $\vee$ Packed_C_F2 ; *Effect*: Packed_C_F1 $\longrightarrow$ billed-Card_F1, Packed_C_F2 $\longrightarrow$ billed-Card_F2
Name: Bill-Cash-C (**S5**) *Input*: Packed_C_F1, Packed_C_F2 *Output*: billed-Cash_F1, billed-Cash_F2 *Precon*: Packed_C_F1 $\vee$ Packed_C_F2 *Effect*: Packed_C_F1 $\longrightarrow$ billed-Cash_F1, Packed_C_F2 $\longrightarrow$ billed-Cash_F2	*Name*: Bill-Card-A (**S6**) *Input*: Packed_A_F1 *Output*: billed-Card_F1 *Precon*: Packed_A_F1 *Effect*: Packed_A_F1 $\longrightarrow$ billed-Card_F1
Name: Bill-Cash-A (**S7**) *Input*: Packed_A_F1 ; *Output*: billed-Card_F1 *Precon*: Packed_A_F1 *Effect*: Packed_A_F1 $\longrightarrow$ billed-Card_F1	*Name*: Bill-Card-B (**S8**) *Input*: Packed_B_F2 ; *Output*: billed-Card_F2 *Precon*: Packed_B_F2 *Effect*: Packed_B_F2 $\longrightarrow$ billed-Card_F2
Name: Bill-Cash-B (**S9**) *Input*: Packed_B_F2 *Output*: billed-Card_F2 *Precon*: Packed_B_F2 *Effect*: Packed_B_F2 $\longrightarrow$ billed-Card_F2	*Name*: Dispatch Service (**S10**) *Input*: billed-Cash F1, billed-Cash F2, billed-Card F1, billed-Card F2 *Output*: dispatch_F1, dispatch_F2 *Precon*: (billed-Cash F1 $\vee$ billed-Card F1 $\vee$ billed-Cash F2 $\vee$ billed-Card F2) *Effect*: (billed-Cash F1 $\vee$ billed-Card F1) $\longrightarrow$ dispatch_F1, (billed-Cash F2 $\vee$ billed-Card F2) $\longrightarrow$ dispatch_F2

Table 1. *Action Specification for the Example problem.*

Definition 8 *(Sound) Plan: A potential plan is a sound plan if each action in the plan, when applied in order to the initial state, will lead to a state which encompasses the goal state.*

Definition 9 *(Sound) Contingent Plan: A potential contingent plan is sound if each branch of the contingent plan, when individually applied to BS_I, will lead to a state which will encompass at least one WS in the goal state.*

Definition 10 *UserAcceptableContingent Plan: A contingent plan is user acceptable if the function* UserSpecCompatible *returns $TRUE$ on the plan and for each branch where the goal is reachable, the plan fragment is Sound-Contingent.*

We allow for planning to be performed when the domain is still incompletely modeled. As a result, there may be conditions under which the goal is *UserAcceptableContingent* but not sound. We require an acceptable plan to be sound for the conditions where the goals are achievable. *UserSpecCompatible* is a function which takes the user specification as input and returns $TRUE$ if the branches in the plan satisfy the specifications, else it returns false. If the user asks for *UserAcceptableContingentPlan* plan, then for a branch in which the goal is unreachable, we give a partial plan that is closest to the goal. But if the user only wants sound plans, a plan is not returned until it is sound along all the branches for which plan (fragments) are generated.

The branches of a contingent plan can be characterized by two properties in addition to the branch conditions.

Definition 11 *Branch Characteristics: A branch of CP is characterized by -*

- *Ambiguity: We measure the ambiguity of a branch by the number of sensing actions in its plan fragment. Since sensing actions represent resolution of uncertainty, the more ambiguous the branch is before execution, the more sensing actions it will have.*
- *Length: It is the total number of actions (sensing + non-sensing) in a branch.*

A ContingentPlan corresponding to the ExampleProblem is given in Figure 1. Note that in this plan, for each branch in which goal is reachable, the plan fragment is sound. The sound branches are numbered in black while the unsound ones are in gray. In the plan, the actions with their initials 'Sense' are sensing actions and hence there are two branches arising as a result of the application of these actions. There are more than one action at the leaf node of the branches, they are to be applied sequentially in the order in which they are written.

2.1 Key User Preferences

The function *UserSpecCompatible* takes user inputs related to preferred plans and checks whether they are satisfied for a given (partial) plan. The current version of the planner supports these inputs:

- Partial branch condition specification (ϕ)
- Number of branches (K) in the plan
- Relative action cost (RC)

Partial branch specification (ϕ): It is a a boolean formula on L_O. For instance, if $L_O = \{P, Q, R\}$, then $\phi = (\neg P \wedge Q) \vee (P \wedge \neg Q)$ can be a branch preference. It means that the branches of interest are the ones in which either P is FALSE and Q is TRUE or P is TRUE and Q is FALSE. The values of rest of the elements of L_O (R in this case) are don't care. The specification is considered partial because it does not say anything about the branches which are outside the specified conditions (R or ¬R in this case). Following are the rules to check if a belief state BS satisfies a single literal specification, l_i. ϕ is checked similarly in a straight forward manner[2].

[2]The default value of ϕ is *TRUE* which means that all states satisfy ϕ.

Rule 1: $l_i \in BS \rightarrow BS$ satisfies l_i.
Rule 2: *unknown*-$l_i \in BS \rightarrow BS$ satisfies both l_i and $\neg l_i$.
Rule 3: Otherwise, BS satisfies $\neg l_i$.

Number of Branches (K): This is the total number of branches which the user wants the planner to find [3].

Relative action cost (RC): The user can set this value to express his preference for characteristic of the plan: shorter in length branches or less ambiguous (less number of sensing actions) branches. By setting a higher value to this parameter the user can increase the cost of sensing action and thereby discourage the selection of the branch which has more sensing actions and vice versa [4].

3 Solving the Planning Problem

In this section, we describe how the user inputs are used to find *UserAcceptableContingentPlan*s. We follow A* search, so at each step, we define cost of any belief state BS by:

$$f(BS) = g(BS) + h(BS)$$

In this formula, g is the cost to achieve the current node and is calculated as the number of actions which were applied to BS_I to reach the state BS and h is the heuristic value. At each time in search, the state with minimum f value is selected. We search in the forward direction and prefer heuristic computation in the backward direction since approximate distance from BS_G is to be computed. We now describe how we calculate the h value and then in search sub-section, we will show how we use it.

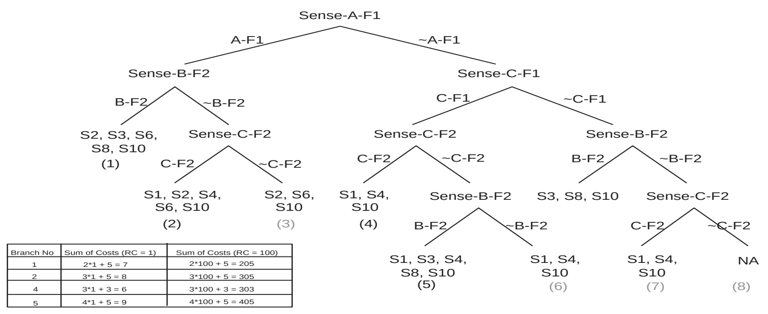
3.1 Calculating Heuristics

The search space of CP is an AND-OR graph of belief states where the sensing actions induce the AND sub-part and the non-sensing actions induce the OR sub-part of the graph. We introduce a cost measure to estimate the heuristic distance in the AND-part which helps in pruning search space based on user specifications. For the OR-part, we could use any distance-based heuristic and we use the well known Planning Graph (PG) heuristics[7]. The latter is not discussed further.

Cost Assignment to literals: The cost assigned to a literal is the minimum estimate of the sum of the cost of the actions which are required to achieve this literal from goal state when planning regressively. We use the following formulas to calculate the cost:

[3]The default value for this parameter is $2^{|L_O|}$ which means all possible branches.

[4]The default value for this parameter is 1 which means equal preference for sensing and non-sensing actions.



Branch No	Sum of Costs (RC = 1)	Sum of Costs (RC = 100)
1	2*1 + 5 = 7	2*100 + 5 = 205
2	3*1 + 5 = 8	3*100 + 5 = 305
4	3*1 + 3 = 6	3*100 + 3 = 303
5	4*1 + 5 = 9	4*100 + 5 = 405

Figure 1. An Example Plan.

$Cost(l_i) = 0$ if $l_i \in BS_G$, otherwise
$= min(Cost(A_i) + min(Cost(A_i^{Add})))$ (1)
$Cost(A_i) = RC$ if $A_i \in A_S$
$= 1$ if $A_i \in A_{NS}$ (2)

In formula 1, if a literal $l_i \in BS_G$ then its cost is 0 else, it is equal to sum of cost of supporting action action A_i ($l_i \in A_i^{Pre}$) and minimum cost among the $l_i^{\prime}s \in A_i^{Add}$. Then a minimum is taken over the complete formula because there can be more than one supporting actions for a literal and we are interested in finding the minimum estimate. In formula 2, all sensing actions are assigned cost equal to RC and non-sensing ones are assigned cost equal to 1. Intuitively, it can be seen that the formula 1 will give an estimate of the sum of costs of actions which are required to obtain l_i in backward planning. This process of finding cost of $l_i^{\prime}s$ has to be done for each l_i and has to be repeated till there is no change in the cost of any l_i. Once we have done the cost assignments, we can calculate h(BS) for any state as the sum of the cost of all literals true in BS:

$$h(BS) = \sum Cost(l_i) \mid l_i \in BS$$

Our approach is complementary to the existing utility based approaches in which a utility function is provided and the aim is to find the plan with maximum utility. We show how both of these can be combined. Let us take an example utility function, $\hat{h}$(BS). Note that f(BS) is the estimated cost of a state BS and since, cost and utility are opposite in nature because less the cost, more preferred the state is and more the utility, more preferred the state is. Hence, the new cost of state can be defined as:

f(BS) = g(BS) + h(BS) + $\ell *$ $1/\hat{h}$(BS), where

ℓ is the weight which can be adjusted according to the problem setting.

3.2 Search

Once we are done with preprocessing, we start the search procedure.

Data Structures: We maintain one global queue and two lists (Figure 2):

1. *GlobalQueue*: contains states arising from the application of a sensing action which satisfy ϕ.
2. *IncompleteBranchesList*: contains states from which there is no path to goal.
3. *GoalBranchesList*: contains states which encompass BS_G, which are used to retrieve plan to reach these states.

Search Algorithm: The main search routine (Figure 2) takes BS_I, BS_G, A_S, A_{NS} and L_O as input. In Steps 2-3, the termination check is done to see if the #ofGoalsFound are more than the user supplied K or the $GlobalQueue$ is empty. If not, in Step 4, we extract a node from the GlobalQueue. If the extracted state *entails* (contains) the goal state, the counter #ofGoalsFound is increased. In Steps 9-17, the non-sensing actions are first attempted and if no more applicable, the sensing actions are tried in routine *applySensingActions* depending on ϕ.

For searching in the OR part of CP's belief states, we enumerate all $A_i \in A_{NS}$ and choose the one which leads to the state with minimum f(BS) value. Here, h value is calculated with PG heuristics.If there are more than one action with minimum f value, then we use soft constraints to break the tie. An instance of it is mentioned in next section. We also have a cycle check mechanism to ensure that redundant search paths are detected early. We first try to achieve sub-goals with non-sensing actions and if no $A_i \in A_{NS}$ is applicable, *applySensingActions* is called. Here we search in the AND part of the CP's AND-OR graph. Among $A_i \in A_S$, we select the action(s) A' whose atleast one of resultant states $BS_1^{\prime}$ and $BS_2^{\prime}$ satisfy ϕ If more than one such action exists, we select the action which has minimum $f(BS_1^{\prime}) + f(BS_2^{\prime})$. Here for calculating f(BS), we use the cost assigned to literals to calculate the h value. If we cannot find a unique action by this, then we again use soft constrains to break the tie.

```
$Function$ : startSearch
$Inputs : BS_I, BS_G, A_S, A_{NS}, L_O$
$Output$ : UserAcceptableContingentPlan
$GlobalVariables$ : NoOfSlns, GlobalQueue,
                    GoalBranchesList, IncompleteBranchesList.
$LocalVariables : ActionList, S_{Temp}, S_{New}$
01. Begin
02.     GlobalQueue.enque($BS_I$)
03.     Begin while (NoOfSlns $\leq$ K $\wedge$
            $\neg$ GlobalQueue.isEmpty()
04.         $S_{Temp}$ = GlobalQueue.deque()
05.     If goalsSatisfied($S_{Temp}$, $BS_G$)
06.         NoOfSlns++
07.         GoalBranchesList.add($S_{Temp}$), continue;
08.     End if
09.         ActionList = $\{A_i \mid A_i \in A_{NS} \cap$
10.         $preconditionSatisfied(A_i, S_{Temp})\}$
11.         IfActionList.isEmpty()
12.             $applySensingActions(S_{Temp}, A_S)$
13.             continue
14.         For all $A_i \in$ ActionList
15.             $S_{New}$ = applyEffect($S_{Temp}$, $A_i$)
16.             GlobalQueue.enque($S_{New}$)
17.     End while
18. End
```

(a) Main Search Routine

```
$Function$ : applySensingActions
$Inputs : BS_{in}, A_{NS}$
$Output$ : UserAcceptableContingentPlan
$LocalVariables : ActionList, BS_1, BS_2$
01. Begin
02.     ActionList = $\{A_i \mid A_i \in A_S \cap$
        $preconditionSatisfied(A_i, BS_{in})\}$
03.     IfActionList.isEmpty()
04.         IncompleteBranchesList.enque($BS_{in}$)
05.         return
06.     For all $A_i \in$ ActionList
07.         $BS_1, BS_2 = applyEffect(BS_{in}, A_i)$
08.         If $BS_1$ satisfies $\phi$
09.             GlobalQueue.Enque($BS_1$, $A_i$)
10.         If $BS_2$ satisfies $\phi$
11.             GlobalQueue.Enque($BS_2$, $A_i$)
12. End
```

(b) Applying Sensing Action

Figure 2. *Search Algorithm*

Support for incomplete branches: For every branch which is incomplete and satisfies ϕ, of all the reachable states in it, we save the one which has minimum f value [5]. In the end, when the search is complete and the number of branches which satisfy ϕ and lead to goal is less than K, say N. Then we extract *K* - *N* states from the queue, and produce the plan required to reach them as an incomplete branch.

Support for Parallel Plans: While heuristics search planners normally produce sequential plans, P4J-CP can produce plans with parallel actions if the user is interested. We use the techniques of AltAlt-p[13] that presented the idea of online parallelization of sequential plans. We have extended their idea to contingent planning but the parallelism is limited to non-sensing actions.

Support for Scalability: In the web services domain, there are usually very large number of services/actions (in the order of thousands) in the registry that may be irrelevant to a specific composition request. Here, doing an up front *relevancy check* on the actions (i.e., services specification) based on the goal could help in removing the irrelevant actions and hence, cutting down the search space. A method to do it is RIFO[10] which starts from goals and tries to determine relevant actions that can directly support the achievement of goals or indirectly support such relevant actions. We implemented a version of RIFO that additionally takes ϕ into account while computing action relevance. Only relevant actions are used during planning.

Planner Completeness and Soundness: P4J-CP employs A* search and is complete when the user specification ϕ is not provided. If the specifications are given, they are used to prune the search space corresponding to branches which do not satisfy ϕ. As a result, the completeness of search is not guaranteed in this case. P4J-CP can be configured to produce only sound plans or unsound plans. In the case of latter, which is useful while planning in an evolving domain, both sound and unsound branches are generated separately and marked for user's convenience.

[5]*IncompleteBranchQueue* is used to store these states.

ϕ	K	RC	Sound Branches	Output Branches
Nil	1	1	1, 2, 4, 5	4
Nil	1	100	1, 2, 4, 5	1
Nil	2	1	1, 2, 4, 5	1, 4
Nil	2	100	1, 2, 4, 5	1, 4
(C-F1∧ C-F2) ∨(¬C-F1∧¬C-F2)	1	1	1, 2, 4	4
(C-F1∧ C-F2) ∨(¬C-F1∧¬C-F2)	1	100	1, 2, 4	1
(C-F1∧ C-F2) ∨(¬C-F1∧¬C-F2)	2	1	1, 2, 4	1, 4
(C-F1∧ C-F2) ∨(¬C-F1∧¬C-F2)	2	100	1, 2, 4	1, 4

Table 2. Sample Plan Output

4 Applying Contingent Planning to WSCE

We now demonstrate that how the branches produced by P4J-CP vary with the user input. With reference to the *ExampleProblem* and its contingent plan (Figure 1), Table 2 shows the branches produced by our planner corresponding to the few instances of user specification.

This contingent plan has 8 different branches (marked 1 to 8) and among them 4 are sound (1, 2, 4 and 5). For each row, the branch with minimum sum of costs of actions in it(Table 2 and small table in Fig.1) is selected. For first four rows, ϕ is Nil, hence all sound branches satisfy ϕ, whereas for next rows where ϕ is (C-F1∧ C-F2) ∨(¬C-F1∧¬C-F2), every sound branch except 5 satisfy ϕ.

The plan produced has unsound branches: 3, 6, 7, 8. This gives necessary information to the domain modeller regarding the reasons for the plan being unsound. He can either enhance the domain model so that it is taken care of or leave it for the user of the system to insert some default branch for such scenarios.

Instance of Soft Constraint: In the domain, we have two actions for payment, one by cash and other by credit card. Now, for same flower shop, both of these actions have same preconditions. So, when payment is to be made both are applicable. To choose one over other, we use soft constrains. User can add some predicate P in BS_I and write P@Bill-Cash-C (as command line argument), this means the user would prefer to pay in cash over card. We have incorporated such kind of soft constraints and use them to resolve the tie when more than one action leads to minimum heuristic value state.

5 Experimental Results and Discussion

We want to establish if our approach of limited contingency planning by using user specification to control search is efficient. An alternative to our approach would be a naive strategy of finding the complete plan under all conditions, and then filtering the specific branches of user interest. We will show that our approach works significantly better than the naive approach. Second, we want to benchmark our base contingent planning system without any user preference. For this, we compare it to Sensory Graph Plan (SGP)[17] and MBP[4]. While P4J-CP is intended for interactive planning applications, like the web services domain, there are very few standard problems. Hence, we use traditional contingent planning domains. All domains/problems are taken from SGP distribution. All results were taken on a IBM ThinkPad which has 1.6GHz Pentium 4 CPU and 512 MB of RAM running Red Hat Linux V9 on it.

We implemented both the approaches and tested them on various domains. We got identical results across all the domains. Here, we present the results for one of the domain: Bomb in the toilet with one hundred packages (Figure 3). In this domain, the uncertainty is about whether a package has a bomb or not. Corresponding to each package, there is a sensing action, hence there are hundred sensing actions. For naive strategy, both memory and time consumption remain unaffected with number of branches to be generated since it first computes the contingent plan for every possible condition and then does the filtering. Notice that when the number of branches to be generated is low, then our methodology performs significantly better than the naive approach but the difference reduces as the number of branches required increase. This is understandable because when all the branches are required, P4J-CP does not perform any search control and both strategies will work at par.

The comparison with MBP and SGP was done on two domains: bomb in toilet and cassandra (see Table 3[6]). For

Domain & Problem	MBP Time (msecs)	SGP Time(msecs)	P4J-CP Time (msecs)
bt-1sa	5180	60	68
bt-2sa	28470	60	114
bt-3sa	153250	60	132
bt-4sa	NS	70	152
Casandra A1	20	10	52
Casandra A2	20	0	69
Casandra A3	20	-	74
Casandra A4	40	-	101
Casandra A5	10	10	54

Table 3. Comparison of P4J-CP with SGP.

No of Irre-lavant Act.	Time(msecs)(With Relevancy Check)	Time (msecs)(Without Relevancy Check)
100	60	90
500	100	320
1000	140	551
5000	461	7360
10000	601	27800
50000	2383	669883
100000	4246	3521754

Table 4. Significance of Relevancy Check

MBP and P4J-CP, we take sum of preprocessing and search time and for SGP, we take the sum of user non-gc time and system non-gc time. The table shows that the base planning system of P4J-CP (when no user preference is used), is competitive with the current contingent planners.

We also wanted to check if the planner is scalable with adapted RIFO while composing with large number of actions. Table 4 shows this to be indeed the case for P4J-CP. Here, time taken to solve a problem from an extended Bomb-in-the-toilet domain is shown in two scenarios: one, in which the relevancy check was used and the other in which it was not.

Performance of P4J-CP in Web Services Problems: There is no standard benchmark for web services composition which evaluates scalability and usefulness of the compositions returned to the user[7]. We are applying P4J-CP in the context of *Synthy* composition system. Here, planning is performed at the level of web service types and not at the lower level of web service instances. Thus, there is basic scalability in the *Synthy* architecture.

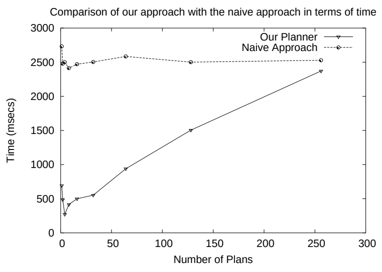
Within *Synthy* , the planner has been used to compose plans in a variety of service scenarios like Helpline Automation[1]. In regular usage, the planner is called to generate plans with up to 3-4 branches and around 20-30 steps which the planner can do in a few seconds. We observe that the planning approach is quite efficient because:

1. User specification is used to effectively cut down on search space.

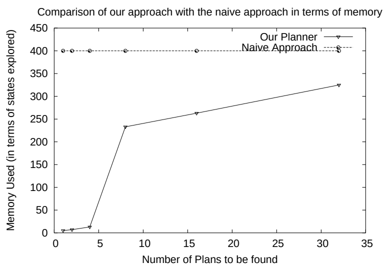
2. The relevance check is effective in reducing the service types (actions) that are considered for planning.

[6]NS indicates not solvable in 500000 msecs. (-) indicates problems not solved by SGP due to memory restrictions of our trial Allegro Lisp version.

[7]There is an attempt at having a syntactic baseline for composition. See http://www.comp.hkbu.edu.hk/ eee05/contest/



(a) Time comparison



(b) Memory comparison.

Figure 3. Our approach against naive approach.

6 Conclusion and Future Work

We have presented a user-driven search control methodology for contingent planning which takes input from user and then uses these inputs to efficiently focus the search. This approach would be specially useful in interactive applications of contingent planning where the response time of the planner is important and the user has preferential interests in different parts of the plan. One such application we are pursuing is in a web services composition tool to build applications. In future, we would like to do more user experiments on the system and integrate the branch specification language with preference languages like CP-Net[3].

References

[1] V. Agarwal, K. Dasgupta, N. Karnik, A. Kumar, A. Kundu, S. Mittal, and B. Srivastava. A Service Creation Environment based on End to End Composition of Web Services. In *Proc. 14th WWW*, May 2005.

[2] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *AI*, 116(1-2):123–191, 2000.

[3] H. Hoos C. Boutilier, R. Brafman and D. Poole. Reasoning with conditional ceteris paribus preference statements. In *Proc. 15th UAI*, 1999.

[4] A. Cimatti, M. Roveri, and P. Bertoli. Conformant planning via symbolic model checking and heuristic search. *Artif. Intell.*, 159(1-2):127–206, 2004.

[5] Y. Lesperance G. De Giacomo and H. J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *AI*, 121(1-2):109–169, 2000.

[6] U. Dal Lago, M. Pistore, and P. Traverso. Planning with a language for extended goals. In *Proc. AAAI/IAAI, Edmonton, Canada.*, 2002.

[7] D. Nau M. Ghallab and P. Traverso. Automated planning : Theory & practice. In *Morgan Kaufmann Publishers, ISBN 1-55860-856-7*, May 2004.

[8] D. McDermott. Estimated-Regression Planning for Interactions with Web Services. In *Proc. AIPS*, 2002.

[9] N. Meuleau and D. Smith. Optimal limited contingency planning. In *ICAPS - Workshop on Planning under Uncertainty and Incomplete Information*, 2003.

[10] Bernhard Nebel, Yannis Dimopoulos, and Jana Koehler. Ignoring irrelevant facts and operators in plan generation. In *Proc. ECP, pages 338-350.*, 1997.

[11] Nilufer Onder and M. E. Pollack. Contingency selection in plan generation. In *Proc. 4th ECP*, pages Pg. 364–376, Toulouse, France, 1997.

[12] R. Petrick and Fahiem Bacchus. A knowledge-Based approach to Planning with Incomplete Information and Sensing. In *Proc. AIPS 2002*, 2002.

[13] R. Sanchez and S. Kambhampati. Altaltp: Online parallelization of plans with heuristic state search. In *In JAIR.*, 2003.

[14] E. Sirin and B. Parsia. Planning for Semantic Web Services. In *Semantic Web Services Workshop at 3rd International Semantic Web Conference*, 2004.

[15] B. Srivastava. A Software Framework for Building Planners. In *Proc. KBCS*, 2004.

[16] B. Srivastava and J. Koehler. Planning with Workflows - An Emerging Paradigm for Web Service Composition. ICAPS 2004 Workshop on Planning and Scheduling for Web and Grid Services, 2004.

[17] Daniel S. Weld, Corin R. Anderson, and David E. Smith. Extending graphplan to handle uncertainty and sensing actions. In *AAAI/IAAI*, pages 897–904, 1998.