Continual Planning with Sensing for Web Service Composition

Eirini Kaldeli and **Alexander Lazovik** and **Marco Aiello**
Distributed Systems Group
Johann Bernoulli Institute
University of Groningen
Nijenborg 9 – 9747AG Groningen
The Netherlands

Abstract

Web Service (WS) domains constitute an application field where automated planning can significantly contribute towards achieving customisable and adaptable compositions. Following the vision of using domain-independent planning and declarative complex goals to generate compositions based on atomic service descriptions, we apply a planning framework based on Constraint Satisfaction techniques to a domain consisting of WSs with diverse functionalities. One of the key requirements of such domains is the ability to address the incomplete knowledge problem, as well as recovering from failures that may occur during execution. We propose an algorithm for interleaving planning, monitoring and execution, where continual planning via altering the CSP is performed, under the light of the feedback acquired at runtime. The system is evaluated against a number of scenarios including real WSs, demonstrating the leverage of situations that can be effectively tackled with respect to previous approaches.

Introduction

The ability to aggregate loosely-coupled software components in order to provide added-value functionalities opens up new prospects for the development of service-oriented applications. Research in the discipline of AI planning can provide deeper insight into the problem of dynamic integration of services, and contribute towards realising an infrastructure that is highly interactive and adaptive to different user preferences. The common premise underlying this approach is that services come along with semantic markups that describe their behaviour in some convenient format, usually in terms of preconditions and effects.

Composition of services is commonly divided into two complementary tasks: *synthesis* or vertical composition, which is concerned with finding the right combination of "abstract" services, each of which models the logic of the provided functionality; and *orchestration* or horizontal composition, which involves instantiating the logical synthesis into concrete WS components, since usually there are many functionally equivalent providers (e.g. hotels, stores, etc.). In this work, we show how the interchange between these two aspects can satisfy a user's request, under variant, unforeseen execution circumstances.

Regarding vertical composition, we propose the deployment of a domain-independent planner to build syntheses automatically and on-demand, relying solely on individual descriptions of decoupled services, and a goal specified by the user. The idea is to maintain a generic and modular repository that comprises a number of diverse service operations, from booking flights to arranging appointments with a doctor, and can serve an variety of different user needs with minimal request-specific configuration. This is different from many previous approaches, which restrict the applicability of the domain to a set of anticipated user needs, predefined in the form of some procedural template, be it in the form of e.g. HTN methods (Au, Kuter, and Nau 2005) or Golog programs (Sohrabi, Prokoshyna, and Mcilraith 2006).

Unlike these approaches, we propose an extended language which allows users to express their goals in a declarative fashion, without having to know about the particularities and interdependencies of the available services. Temporal aspects, maintainability properties, and distinguishing between wish to observe the environment or change it are some of the features this language supports. Both the domain and the goal are modeled as a CSP (Constraint Satisfaction Problem), and a constraint solver is applied to compute a plan. An important advantage of the CSP-based formulation is its efficient handling of variables ranging over large domains, which are very common in WS fields (e.g. dates, prices). Some preliminary ideas about service composition via CSP-based planning with extended goals have been presented in (Kaldeli, Lazovik, and Aiello 2009), where the basic concepts of the synthesis approach are presented.

Due to the conditions of incomplete knowledge and sensing, as well as other sources of contingency (e.g. corrupt responses, server failures etc.), the problem of composition cannot be tackled without reference to the current environmental context, that becomes visible only at execution time. The problem of missing knowledge and non-determinism in the field of WSs has been addressed from different perspectives, although recovery from unforeseen outcomes is generally disregarded. Previous approaches either rely on conditional plans and can therefore handle only a limited range of non-deterministic action outcomes, e.g. (Pistore et al. 2005), (Hoffmann, Weber, and Kraft 2010), or have the queries about unknown information explicitly included in the predefined WS procedure, e.g. (Au, Kuter, and Nau 2005), (Sohrabi, Prokoshyna, and Mcilraith 2006).

In this work, we employ a knowledge-level representation that enables plans to be automatically built based on the agent's knowledge and the way that this is changed by actions. To deal with the large number of possible outcomes and unforeseen contingencies at runtime, we adopt an approach that interweaves planning, monitoring and execution. *Continual planning* is performed, so that the upcoming plan steps anticipated off-line can be revised as execution proceeds, in face of inconsistencies that stem either from the newly acquired information or from service failures. In the algorithm presented in this work, we show how dynamic constraint solving, that allows efficient addition and removal of constraints, can serve the need for constantly incorporating new facts about the environment or removing obsolete ones, checking for inconsistencies, and reacting accordingly.

A motivating example

Let us suppose that a user is happy to learn that in the following days a singer he is fond of is making a tour in the country where he lives. What he wants is to book a ticket and a hotel room for the nearest upcoming concert whose date and location meet some criteria referring to the weather conditions, the distance from his hometown, and his availability according to his agenda, as well as about the price he is willing to pay for his overnight stay. These requirements are expressed by an extended goal (see the third section), that specifies *what* (but not how) the user wants to achieve and under which conditions.

The satisfaction of this goal requires the collaboration of services coming from diverse business domains –namely related to travelling, entertainment events, maps, calendar and weather services– in a manner that can be hardly anticipated in advance. Depending on the information returned at runtime, there are clearly many different ways that this goal can be fulfilled. For example, it may turn out that the place of the first upcoming concert is too far, or that there is no hotel available on that date within his badget, etc. In such cases, the original plan has to be interrupted and revised, so that the conditions regarding the whereabouts and date of the next concert are looked up. To further complicate things, at any moment a service may fail. So, if e.g. the booking service of the first selected hotel that meets the user's criteria happens to be in a permanent failure state, an alternative hotel has to be searched, and depending on the result, the goal may finally be satisfied or not. In the penultimate section, we show a possible run of the continual planning algorithm for this scenario, based on real data received by the services.

Representing the domain

A WS marketplace is conceived as a planning domain, where the actions correspond to operations of abstract WSs. We assume that the service domain description is carried out by a domain designer, who formalises the individual WS operations by providing the necessary markups. How existing Semantic Web ontologies can be exploited in the context of planning is an interesting topic, e.g. OWL-S used in (Au, Kuter, and Nau 2005), however it is not the focus of this paper.

Definition 1 (Service Domain). A service domain is a tuple $\mathcal{SD} = \langle Var,\ Par,\ Act\rangle$, where:

- Var is a set of variables. Each variable $v \in Var$ ranges over a finite domain D^v.
- Par is a set of variables that play the role of input parameters to WS operations. Each variable $p \in Par$ ranges over a finite domain D^p.
- Act is the set of actions. An action $a \in Act$ is a triple $a = (id(a),\ precond(a),\ effects(a))$, where $id(a)$ is a unique identifier, e.g. "bookHotel", $precond(a)$ is a set of propositions on variables and parameters, and $effects(a)$ is a set whose elements can be one of the followings:
  - $sense(var)$, where $var \in Var$
  - $assign(var, v)$, where v is some constant or $v \in Var$
  - $assign(var, f(v_1, v_2))$, where $v_1, v_2 \in Var$ or v_1, v_2 are constants, and f the sum or the subtract function
  - $increase(var, v)$ or $decrease(var, v)$, where $v \in Var$ or v is some constant

A state s is defined as a tuple $s = \langle(x_1, D_s^{x_1}), \ldots, (x_n, D_s^{x_n})\rangle$, where $x_i \in Var \cup Par$ and $D_s^{x_i} \subseteq D^{x_i}$. The domain of x at state s is given by the *state-variable* function $x(s)$, so that $x(s) = D_s^x$ if $(x, D_s^x) \in s$. If $|D_s^x| = 1$, this means that x at s has a specific value. The effects of type $sense(var)$ are called *observational*, i.e. they observe the current value of a variable, while the other types of effects are *world-altering*, i.e. actively change the value of a variable. An action may have both kinds of effects.

The domain is extended by additional variables to model the knowledge level, and to distinguish between sensing and world-altering actions. These variables are generated automatically given a domain description $\mathcal{SD}$. First, for each $var \in Var \cup Par$, a new boolean variable var_known is introduced, which indicates whether var is known at state s ($var\_known(s) = true$) or not ($var\_known(s) = false$). For every variable $kvar \in Var$ that participates in an observational effect, a new variable $kvar\_response$ is created, which is a placeholder for the value returned by the respective sensing operation. Since this value is unknown until execution time, $kvar\_response$ ranges over $kvar$'s domain ($kvar\_response \in D^{kvar}$). Moreover, we maintain for every variable $cvar \in Var$ that is part of at least one world-altering effect a boolean flag $var\_changed$, which becomes true whenever this effect takes place. All these additional variables are created automatically, by parsing the $\mathcal{SD}$. Thus, we end up with an extended set of variables $V = Var \cup Par \cup Kb \cup Cv \cup Rv$, where Kb is the set of knowledge-base variables, Cv the set of the change-indicative variables, and Rv the response variables.

Encoding the domain into a CSP

A constraint satisfaction problem is a triple $CSP = \langle X, \mathcal{D}, \mathcal{C}\rangle$, where $X = \{x_1, \ldots, x_n\}$ is a finite set of n variables, $\mathcal{D} = \{D^1, \ldots, D^n\}$ is the set of finite domains of the variables in X so that $x_i \in D^i$, and $\mathcal{C} = \{c_1, \ldots, c_m\}$ is a finite set of constraints over the variables in X. A constraint c_i involving some subset of variables in X is a proposition that restricts the allowable values of its variables. A solution to a $CSP\ \langle X, \mathcal{D}, \mathcal{C}\rangle$ is an assignment of values to the variables in $X\ \{x_1 = v_1, \ldots, x_n = v_n\}$, with $v_i \in D^i$, that satisfies all constraints in $\mathcal{C}$.

Following a common practice in many planning approaches, we consider a *bounded* planning problem, i.e. we restrict our target to finding a plan of length at most k, for increasing values of k. Considering a service domain extended with the knowledge-level representation $\mathcal{SD}' = \langle V, Act\rangle$, the target is to encode SD' into a $CSP = \langle X_{CSP}, \mathcal{D}, \mathcal{C}\rangle$. First, for each variable $x \in V$ ranging over D^x, and for each $0 \leq i \leq k$, we define a variable $x[i]$ in CSP with domain D^x. Actions are also represented as variables: for each action $a \in Act$ and for each $0 \leq i \leq k{-}1$ a boolean variable $a[i]$ is defined. This way the computed plan can include parallel actions, a fact that may save time during execution. If some action a_1 affects a variable that is part of the preconditions of some other action a_2, or if both affect the same variable, then a_1 and a_2 are prevented from being put in parallel by an additional constraint.

Action preconditions and effects, as well as frame axioms, are automatically encoded as constraints on the CSP state variables. Due to space reasons, we do not give the details of the constraint representation, but rather provide an example of how two simple actions are modelled in the form of constraints.

payIn(amountPar, accIdPar)

`prec:` $\emptyset$ `effects:` $\mathit{increase}(\mathit{accBalance}, \mathit{amountPar})$

$\mapsto$ Constraints:

`prec constraints:` /*parameters known*/

$\mathit{payIn}[i] = 1 \Rightarrow$

$(\mathit{amountPar\_known}[i] = \mathit{true} \wedge \mathit{accIdPar\_known} = \mathit{true})$

`effect constraints:` /*world-altering*/

$\mathit{payIn}[i] = 1 \Rightarrow (\mathit{accBalance\_changed}[i+1] = \mathit{true}$

$\mathit{accBalance}[i+1] = \mathit{accBalance}[i] + \mathit{amountPar}[i])$

findAccBalance(accIdPar)

`prec:` $\emptyset$ `effects:` $\mathit{sense}(\mathit{accBalance})$

$\mapsto$ Constraints:

`prec constraints:` /*parameters known and accBalance yet unknown*/

$\mathit{findAccBalance}[i] = 1 \Rightarrow (\mathit{accIdPar\_known}[i] = \mathit{true} \wedge$

$\mathit{accBalance\_known}[i] = \mathit{false})$

`effect constraints:` /*sensing*/

$\mathit{findAccBalance}[i] = 1 \Rightarrow \mathit{accBalance\_known}[i+1] = \mathit{true} \wedge$

$\mathit{accBalance}[i+1] = \mathit{accBalance\_response}[i+1]$

Extra knowledge preconditions that ensure that the knowledge-base variables of all input parameters should be true are added. It's worth mentioning that most actions in WS domains have only knowledge preconditions, like in the above examples. Effects of the type $sense(var)$, such as in the case of *findAccBalance*, are modelled by adding an extra precondition that the variable to be sensed is unknown (to avoid redundant sensing), assigning to var its corresponding response variable, and setting the respective knowledge variable to true. For each world-altering effect we add an additional constraint that ensures that any variables participating in the second argument of the assignment effect should be already known (in the case of *payIn* this is already guaranteed by the preconditions), and states that the variable altered becomes known.

By adopting such an encoding, the required sensing actions are determined *pro-actively*, depending on the goal and the knowledge the user already possesses. The offline solver may assign arbitrary values to an unknown variable var, however if the corresponding knowledge variable var_known is false, this values is of no validity. The effect of this behaviour is that the planner always generates an optimistic plan, i.e. anticipating that all knowledge-gathering actions return information that is in accordance with the user's requirements, and all actions are executed successfully. This initial plan is revised during execution as we will see in the section about the continual planning algorithm.

The goal language

In (Kaldeli, Lazovik, and Aiello 2009) a language for extended goals in the context of CSP planning for WS compositions is presented. This goal language has been enriched with a few more constructs, so that more complex goals can be formulated. Due to lack of space, we do not repeat the syntax of the language here, or explain the formal semantics, but rather give an impression of some of the supported constructs through the following two examples:

Goal 1

`achieve-maint(`$\mathit{bookedConcert} = \mathit{TRUE}$`) under_condition`
`(find_out-maint(`$\mathit{temperature} > 0$`))`

Goal 2

`achieve-maint(`$\mathit{bookedHotel} = \mathit{TRUE}$`)` $\wedge$ `(`
`achieve-maint(`$\mathit{bookedConcert} = \mathit{TRUE}$`)`
`under_condition_or_not (find_out-maint(`$\mathit{temperature} > 0$`)))`

An `achieve-maint(`$\wedge_i prop_i$`)` subgoal on a conjunction of propositions implies that $\wedge_i prop_i$ has to become true at some state, and stay true till the final one. The `under_condition` structure imposes that Goal 1 is accomplished if s is the first state at which $\mathit{bookedConcert} = \mathit{TRUE}$ is satisfied, and `find_out-maint(`$\mathit{temperature} > 0$`)` is satisfied in the state sequence preceding s. If $\mathit{temperature} < 0$, then Goal 1 fails. `under_condition_or_not` is a new construct that elaborates the expressivity of the language: $goal_0$ `under_condition_or_not` $goal_1$ will also be fulfilled if $goal_1$ is not satisfiable, if however it is, then $goal_0$ has to be as well. Thus, Goal 2 will ensure that $\mathit{bookedConcert} = \mathit{TRUE}$ will be satisfied if the temperature is not below zero, while if it is, then only $\mathit{bookedHotel} = \mathit{TRUE}$ will be looked after. It should be mentioned however that the `under_condition_or_not` structure works as intended only if the variables involved in $goal_1$ are known at planning time. `find_out(`$\wedge_i k\text{-}prop_i$`)` type of subgoals enforce a hands-off requirement on the variables they involve, i.e. the planner will try to satisfy the propositions at some state without allowing any altering effect on these variables (in the above examples `find_out-maint` is in practice unnecessary because there is no way to change the weather).

An example of how the constraint encoding of the goal looks like is provided for *Goal 1*:

$\mathit{bookedConcert}[k] \wedge \mathit{bookedConcert\_known}[k] = \mathit{true}$

```
for i ← 0, k − 1                /*maint constraints*/
 for j ← i + 1, k
   $(bookedConcert[i] \wedge bookedConcert\_known[i]) \Rightarrow$
   $(bookedConcert[j] \wedge bookedConcert\_known[j])$
/*knowledge variables should remain unchanged (find-out goal)*/
$\neg temperature\_changed[k] \wedge temperature\_known[k]$
for i ← 1, k               /*under_condition goal*/
 $bookedConcert[i]) \Rightarrow$
   $(temperature[i-1] > 0 \wedge temperature\_known[i-1])$
```

All variables and parameters not specified in the goal are assumed to be undefined, and their respective knowledge-level variables are set to false. Functions as part of the goal are also allowed, and are translated to a set of propositions, and this is how parameters are mapped. Thus, in the last example, the user would rather specify $bookedHotel(hPlacePar, hDatePar)$, where $hPlacePar$ and $hDatePar$ can be either a specific value (e.g. $hPlcacePar =$"*Freiburg*") or refer to some other variable, that may correspond to the yet unknown outcome of some other action (e.g. $hDatePar = eventDate$). This amounts to assigning the respective parameter to the provided value at the initial state, or fix it to the specified variable at all states.

Orchestration by Continual Planning

After the invocation of the solver, an assignment to the action variables is returned, which corresponds to an optimistic plan, as already mentioned. Before resorting to the solver, a preliminary pruning of the actions that are irrelevant to the goal is performed. This initial step is important to avoid redundant sensing or unwanted world-altering actions, and also contributes to enhancing scalablity, since, given a large set of WSs, usually only a minority is relevant to a particular goal. The off-line plan is then passed to the orchestrator, whose task is to gradually update it, according to the information it acquires from the actual physical service invocations. The ultimately successful plan–if one exists–is constructed step-by-step, by adding and removing constraints from the constraint network. Since the off-line plan has no way to anticipate any value that is to be observed, whenever new information is sensed, some revision is needed. For example, if the user wants to send some mail to a particular address, which is unknown and has to be supplied by some address-providing service, then at the point when the address becomes known, re-solving is required to instantiate the right input arguments of the sensing actions that depend on that information.

The orchestrating algorithm relies on the reasonable information persistence assumption, i.e. that the knowledge collected by the actions at run-time remains valid till the end of the algorithm. can also be seen as a form of incomplete knowledge which require special handling. The algorithm accommodates for simple types of flaws, assuming that all failures are clean, i.e. either all world-altering effects of an action are materialised, or none of them is. Erratic situations can only be avoided, if actions with potentially severe world-altering effects, that e.g. involve a payment, are reversible. The algorithm for orchestrating and adjusting the initial plan proceeds as outlined in the pseudocode, while a running example follows. In the description provided herein, the generated plans are assumed to be serialized, because no support for concurrent service calls is implemented yet.

Algorithm: Continual Planning

```
function ORCHESTRATE($plan$)
  for state $s_i \in plan$ do
    repeat
      $output$ = NEXT_INSTANCES($act_i, inParams(act_i, s_i)$);
      $good$ = CHECK_VIOLATION($output, s_i, act_i, instance$);
    until output indicates flaw $OR$ no more instances $OR$ $good$
    if output indicates flaw then
      Forbid $act_i$ (with same input); BACKTRACK;
    else if no more instances then
      Add constraints about $act_i$'s inspected outputs;
    if no more instances $OR$ $\neg good$ then
      BACKTRACK; return ;
function CHECK_VIOLATION($output, s_i, act_i, instance$)
  Bookkeeping information for $output$;
  Check if $output$ at $s_{i+1}$ causes conflicts;
  if no conflict then
    Push $\{s_i, act_i, instance\}$ to backtracking stack;
    Form $newInitState$ by materialising $act_i$'s effects;
    Compute $newPlan$ from $newInitState$; return true;
  else return $false$;
function BACKTRACK
  if backtracking stack not empty then
    Pop $\{s_i, act_i, instance\}$ from backtracking stack;
    if backtrack due to violation and $act_i$ had severe
    committing effects then   Undo $act_i$'s effects;
    Update alternative instances for actions;
    Compute $newPlan$ from $s_i$;
    if $newPlan$ is found then   ORCHESTRATE($newPlan$);
    else  {BACKTRACK; return ; }
  else
    Compute $newPlan$ from original initial state;
    if $newPlan$ is found then   ORCHESTRATE($newPlan$);
    else  return fail;       /*The goal cannot be satisfied*/
```

Let us now explain the algorithm step-by-step. For each action in the plan, a list of physical instances that match its functionality is kept. NEXT_INSTANCES is responsible for selecting the next alternative concrete service that matches the logical action and execute it. The matchmaking process may be based on Quality of Service metrics, or also take into account user-specific preferences, e.g. see (Skoutas et al. 2008), and is not the focus of this work. NEXT_INSTANCES goes on with executing the next available instance, till it finds a service which returns no failure. If no such service can be found, the algorithm starts backtracking from the previous state, to look for alternative plans, after adding a constraint that forbids the action in question to be chosen by subsequent plans. Moreover, in case the output information of all meaningful instances has been collected, following the information persistence assumption, the algorithm incorporates in the constraint network the knowledge that whenever the respective action is invoked with the same arguments, it returns one of the already sensed values.

In case of a flawless invocation that returns some new information, the CHECK_VIOLATION function is called to inspect whether this output violates any constraints. World-altering effects don't have to be checked, since they are already reasoned about at planning time. If a violation is detected, then the algorithm goes on with trying an alternative physical service. For services whose output may differ depending on the selected provider, such as stores returning the availability or price of a requested item, it makes sense to try alternative instances, while this is not the case for services that provide information that is not instance-specific, such as the weather, map etc. If all alternatives prove unsuccessful, the algorithm tries to backtrack, otherwise the current state of the world is recorded for prospective backtracking, the planner computes an updated plan $newPlan$, and proceeds with executing it. The updated plan is computed after disregarding the previous solution (assignments to variables), and by considering as the initial state a new state $newInitState$, which reflects the state after the materialisation of act_i's effects. This $newInitState$ is constructed based on the propagation of constraints entailed by act_i, i.e. act_i's effects, both observational and world-altering, and the instantiation of relevant input parameters depending on them (which were previously assigned to some arbitrary convenient values). All variables for which the knowledge base indicates they are unknown are restored to undefined.

When backtracking from $\{s_i, act_i, instance\}$, and act_i has lead to the materialisation of some severe world-altering effects, then these should be undone. Note that the majority of services are usually purely information-providing, so reversal of effects at this stage is rarely necessary. Then, all possible instances for all other actions except act_i are restored. The algorithm generates a new plan, under the light of the new constraints, following the same procedure as described above. If it fails, then it backtracks to the previous stored world state, and the same process is recursively repeated until either a new plan is found, or the initial state is reached.

A running example

In the followings, we show a running instance of the orchestration algorithm for the motivating example described in the introduction, with a user living in Stanford, CA, who wants to attend a concert of the singer Tina Dico.

Initial plan: {getFirstEvent(Tina Dico), checkCalendarAvail(defaultDate), getDistance(Stanford, defaultPlace), getTemperature(defaultPlace, defaultDate), bookConcertTicket(Tina Dico, defaultDate, defaultPlace), search4Hotel(defaultDate, defaultPlace, 1, 1), bookHotel(hotelWS, defaultDate, defaultPlace, 1, 1)}

$\rightarrow$ Call getFirstEvent(Tina Dico) :: eventDate=2011-02-05, eventPlace=Austin
$\rightarrow$ Call checkCalendarAvail(2011-02-05) :: calendarAvail=true
$\rightarrow$ Call getDistance(Stanford, Austin) :: distance=2793
Sensed value 2793 for distance violates constraints, **Backtrack**
No alternative meaningful instances for checkCalendarAvail, **Backtrack**
$\rightarrow$ Call getSecondEvent(Tina Dico) :: eventDate=2011-02-08, eventPlace=San Francisco
$\rightarrow$ Call checkCalendarAvail(2011-02-08) :: calendarAvail=true
$\rightarrow$ Call getDistance(Stanford, San Francisco) :: distance=62
$\rightarrow$ Call getTemperature(San Francisco, 08 Feb 2011) ::temperature=11
$\rightarrow$ Call bookConcertTicket(San Francisco, 2011-02-08) :: bookedConcTicket=true
$\rightarrow$ Call search4HotelA(2011-02-08, San Francisco, 1, 1) :: hotelWS=Chancellor Hotel, hotelPrice=80
$\rightarrow$ Call bookHotel(Chancellor Hotel, San Francisco, 1, 1) :: null
A failure occurred, **Backtrack**
$\rightarrow$ Call search4HotelB(2011-02-08, San Francisco, 1, 1) :: hotelWS=Fairmont Hotel, hotelPrice=100
$\rightarrow$ Call bookHotel(Fairmont Hotel, San Francisco, 2011-02-08, 1, 1) :: hotelBooked=true

All services in this example, except the ticket and hotel-booking ones, are real services available on the Web (so, Tina Dico was indeed performing on the mentioned places and dates, etc.). The Yahoo! weather service provides information such as the temperature or the weather condition, e.g. "rainy" or "cloudy". Google calendar is used to check whether a day is marked free or busy, to find the distance between two locations. The `eventful.com` service provides information about a number of cultural events, and in this example it is used to go through the list of concerts of a given band. The responses of the actual services are XML documents, which are parsed to extract the respective information. Because the Yahoo! weather-related services require WOEIDs (Where on Earth IDentifier) as the form of their location-related input parameters, an intermediate service operation call is performed to map the location names to this format. Dates are also transformed between different formats, depending on the specification of each service. Notice also that the policy for dealing with $bookHotel$'s failure response in this case is to consider it a permanent one.again with the same input arguments. Evidence about the time performance of the algorithm is summarised in the third row, test `[3a]` of Table 1. It should be mentioned that out of the 18.6 sec of total execution time, 6 sec amount to the sum of the service calls response times.

Empirical Evaluation

The aim of the evaluation scenarios for the planning and orchestration framework is to test whether complex goals can be accomplished within acceptable time under different environmental context, i.e. sensed outcomes and failures. The tests involve a mixture of real and virtual services, derived from a variety of different application domains: making online appointments, shopping, shipping, travelling, learning about entertainment events, and obtaining general purpose information, e.g. from maps or weather services. In total, the domain consists of 30 abstract service operators (it should be emphasised that these refer to ungrounded operators), 23 of which are knowledge-providing. 45 service instances are used for the test purposes, however it should be noted that scalability regarding the number of physical components depends on the efficiency of the matchmaking process. To experiment with failures, we have simulated "return null" responses to model simple flaws. The experiments were performed on an Intel Core i5 2.26Ghz computer with 3GB of RAM, running Java 1.6.0_12. The constraint solver standing at the core of the planner is the Choco v2.1.1 constraint solving library (`www.emn.fr/x-info/choco-solver`). The bootstrap time for loading the domain description and

Goal	♯ actions in initial plan	Initial plan time	Test instance	Total time	♯backtracks/ ♯violation checks
appointment	5	0.9	`[1a]`	11.9	1/5
select&buyCd	6	2.1	`[2a]`	20.6	0/6
arrangeGoToConcert	7	2.8	`[3a]`	18.6	3/9
			`[3b]`	164.7	12/16
buyBook&ship	9	4.3	`[4a]`	32.7	3/7
arrangeTravel	12	4.6	`[5a]`	27.5	0/8
			`[5b]`	225	19/22
			`[5c]`	344.3	23/30
combinedGoal	19	7.1	`[6a]`	216.3	1/16

Table 1: Results for different goals and execution circumstances (time in sec). The tests correspond to runs for the same goal and initial state, but different returned outputs and failures. Total CPU time counts the time elapse between issuing the goal and its satisfaction or failure.

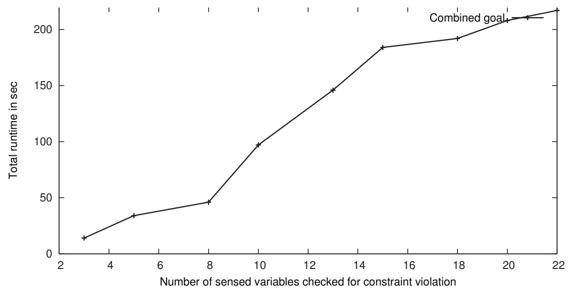

Figure 1: Total runtime in sec vs. the number of initially unknown variables which have to be sensed and checked for violation. Measurements are taken for a conjunctive goal tested towards different initial states reflecting a decreasing number of initially known variables.

translating it into constraints is 3.2 sec.

The results of running a number of diverse scenarios are summarised in Table 1. Each goal's fulfillment requires a different combination of services coming from the various business fields covered by the domain, and for a given initial state, an off-line plan is generated. The test instances correspond to variant runs of the orchestrator for this initial plan, depending on the different feedback received at execution. The amount of backtracking is dependent on the returned outputs, as well as on the order of invocations instructed by the planner, while the response times experienced by the same service may differ considerably at different invocations. The reported times are the average over 3 separate runs of the same test instance. It should be emphasised that the test instances indicated by `b` or `c` are deliberately modeled for experimenting with the orchestrator's behaviour under extremely ill-behaved circumstances, where alternative service instances consecutively fail to satisfy the goal, e.g. all instances promise to provide the desired output, i.e. they have a room, car etc. available, but at the last moment the booking process fails. The time required for a violation check depends on the constraints entailed by the new information that is gathered.

The overall execution time is dominated by the time spent on inconsistency inspections and re-planning according to the feedback received at runtime. Therefore, in Figure 1 we plot the relation between total runtime and the number of initially unknown variables that have to be sensed, and which entail solving the new CSP instance. The goal used for these tests is an artificial one, deliberately constructed so as to require the invocation of all knowledge-gathering actions under minimal initial knowledge. We see that the algorithm can successfully solve problems with a high number of unknown variables, some of which are of high cardinality. Although no direct comparison with other planning approaches to WS composition can be made, given that they use distinct testing domains and base on different assumptions and aims, it is worth mentioning that in (Au, Kuter, and Nau 2005) only up to 9 unknowns can be dealt with, requiring 100 sec. With respect to off-line synthesis time, the CSP-based planner used herein is slower than FF employed in (Hoffmann, Weber, and Kraft 2010) or SHOP2 in (Au, Kuter, and Nau 2005), however it supports complex goals, variables ranging over very large domains, and re-planning for dealing with dynamic context.

Concluding remarks

We have designed, implemented and evaluated a planning framework for generating automatic WS syntheses, that accommodates for complex goals, a knowledge-level representation to model lack of information and proactive sensing in presence of variables that range over large domains, as well as an algorithm for monitoring execution and revising plans in a dynamic environment. These features put together enhance the extent of scenarios that can be represented and dealt with compared to previous approaches. Experimental evaluation confirms that the framework performs well in different situations, with complex goals, real services, and several combinations of unknowns and failure occurrences.

References

Au, T.; Kuter, U.; and Nau, D. 2005. Web Service Composition with Volatile Information. In *Int. Semantic Web Conf. (ISWC'05)*.

Hoffmann, J.; Weber, I.; and Kraft, F. 2010. SAP Speaks PDDL. In *4th National Conf. of the American Association for Artificial Intelligence (AAAI'10)*.

Kaldeli, E.; Lazovik, A.; and Aiello, M. 2009. Extended Goals for Composing Services. In *19th Int. Conf. on Automated Planning and Scheduling*.

Pistore, M.; Marconi, A.; Bertoli, P.; and Traverso, P. 2005. Automated Composition of Web Services by Planning at the Knowledge Level. In *19th Int. Joint Conference on Artificial Intelligence*.

Skoutas, D.; Sacharidis, D.; Simitsis, A.; and Sellis, T. 2008. Serving the Sky: Discovering and Selecting Semantic Web Services through Dynamic Skyline Queries. In *2nd IEEE Int. Conf. on Semantic Computing*.

Sohrabi, S.; Prokoshyna, N.; and Mcilraith, S. A. 2006. Web Service Composition via Generic Procedures and Customizing User Preferences. In *Int. Semantic Web Conf.*