Extended Goals for Composing Services

Eirini Kaldeli and **Alexander Lazovik** and **Marco Aiello**
Distributed Systems Group
Dep. of Mathematics and Computing Science
University of Groningen
Nijenborg 9 – 9747AG Groningen
The Netherlands

Abstract

The ability to automatically compose Web Services is critical for realising more complex functionalities. Several proposals to use automated planning to deal with the problem of service composition have been recently made. We present an approach, based on modelling the problem as a CSP (Constraint Satisfaction Problem), that accommodates for the use of numeric variables, sensing and incomplete knowledge. We introduce a language for expressing extended goals, equipped with temporal constructs, maintainability properties, and an explicit distinction between sensing and achievement goals, in order to avoid undesirable situations.

Introduction

Automatic Web Service Composition (WSC) addresses the problem of the on-demand combination of loosely-coupled service operations, in order to realise some complex objective, specified by an end-user. AI planning offers suitable tools for achieving such a dynamic composition. In this respect, the operations provided by the Web Services (WSs) available in a business domain are viewed as actions, described in terms of preconditions and effects. WSC problems are characterised by a number of requirements, that distinguish them from classical planning domains, such as the need to deal with incomplete information, numerical fluents, extended goals and non-determinism during execution.

Herein, we propose a modelling of the domain and goal via constraints, and resort to a standard constraint solver to compute a valid plan. Our approach is driven by the aim of maintaining a framework where the business domain is as generic as possible, and plans are built dynamically according to the preferences of the user. This is unlike most previous approaches that rely on pre-defined business processes, which dictate the allowed sequences between the WS operations, like the state transition diagrams in (Lazovik, Aiello, and Gennari 2005) or the method lists required in (Kuter et al. 2004). Our approach accommodates for incomplete knowledge and supports proactive information gathering, i.e. the planner detects in which cases it lacks knowledge and chooses the necessary sensing operations.

Another contribution of this work is the proposal of a rich goal language, that allows for the specification of extended goals, beyond a mere description of the final state. Such an expressive power is missing from other domain-independent approaches to WSC, like (Klusch and Gerber 2006). Although complex goals have been addressed in a number of planning approaches, we are not aware of any such extensions in the context of CSP for planning, with the exception of the work in (Lazovik, Aiello, and Gennari 2005), which is however confined by a rather restrictive domain theory.

Despite the fact that our work is inspired by the requirements set forth by the field of WSs, the applicability of our framework is domain-independent, and touches on many issues that are of concern to the planning community. Thus, it can be deployed in any problem where versatility of domain description, incomplete knowledge, and expressivity of the goal language are at stake.

Representing the domain

A service marketplace is conceived as a planning domain, where the actions correspond to operations of abstract WSs, that may be realised by a number of concrete service providers supplying equivalent functionalities. The service domain description is carried out by a domain designer, who provides the necessary markups for the individual WSs.

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, i.e. "book_hotel", $precond(a)$ is a set of propositions on variables and parameters, and $effects(a)$ is a set of assignments to variables in Var.

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 an action are either *world-altering*, which actively change the value of a variable, or *knowledge-providing*, which sense the current value of a variable. Variables involved in some knowledge-providing effect of an

action are called *knowledge variables*. An action may have both knowledge-providing and world-altering effects. To capture incomplete knowledge, we introduce for every knowledge variable and for every parameter var a new boolean variable var_known, which is set to true whenever var becomes known after a sensing action. A state-variable function is defined for each of these knowledge base variables. At the initial state s_0, $var\_known(s_0)$ may be true or false, depending on whether the corresponding var has a specific value or not. Knowledge-providing effects are of the form $k\text{-}var := k\text{-}response$, where $k\text{-}var$ is a knowledge variable and $k\text{-}response$ is a placeholder for the value returned by the respective sensing operation. Since this value is unknown until execution time, $k\text{-}response$ ranges over $k\text{-}var$'s domain ($k\text{-}response \in D^{k\text{-}var}$).

World altering effects are of the form $var := value$, where variable $var \in Var$ is assigned a value compliant with var's domain, or $var := var \diamond value$, where $\diamond \in \{+, -\}$. We also maintain for every variable $var \in Var$ that participates in at least one world-altering effect a boolean flag $var\_changed$, which becomes true whenever a world altering effect changes the value of var. A state-variable function is defined for the $var\_changed$ variables as well.

Under the light of these supplements, the Var set of the service domain is extended to also include the set of knowledge base variables, the change-indicative flag variables, and the response variables. We call this extended set V.

An example

The travel WS domain comprises a set of providers which offer a number of operations that are useful for organising a trip, and is one of the most established test cases in the WSC literature. In Figure 1 a possible high-level encoding of the operations provided by an air-tickets reservation WS is shown. In this example, we assume that all parameters regarding the dates, origin etc. are specified in advance by the user. The "searchForFlight" (abstract) operation is issued simultaneously to all or some of the respective concrete WSs, and only one particular instance is returned (`flightWS_response`) following some criteria, e.g. the cheapest air-ticket found. It is also an indispensable part of the preconditions of all actions that the respective parameters are known (left implicit in Figure 1).

flight WS

```
searchForFlight (params: fdate1, fdate2,
 origin, destin)::
 prec: flightWS_known=false
 effects: (flightWS:=flightWS_response AND
 flightWS_known:=true) AND
 (flightPrice:=flightPrice_response
 AND flightPrice_known:=true)
bookFlight (params: fdate1, fdate2, origin,
  destin, vegMeal)::
 prec: flightWS_known=true
 effects: flightBooked:=true AND
 flightBooked_changed:=true
```

Figure 1: Booking flights Web Service operations.

Planning by using CSP

Constraint satisfaction has become a powerful solving technique that has been applied to planning either by directly modelling the planning problem as a CSP, e.g. in (van Beek and Chen 1999), or indirectly, e.g. in (Do and Kambhampati 2000). We follow the former approach, proposing a way to represent the planning problem in the form of constraints, which maintain the structure of the high-level domain description.

Planning as constraint satisfaction has a number of advantages, some of which are of particular interest to WS environments. Firstly, it supports efficient handling of numeric variables, which are common in WS interactions. Secondly, suitable fine-tuned variable and value ordering heuristics, as well as different propagation techniques can be applied to speed up solving. Concerning the maintenance of rich goal descriptions, which is a major requirement in a versatile domain, we show that constraints can serve the need for expressing complex goals. A constraint-based system is also well-suited for the interaction between off-line planning and execution, an important aspect that although not addressed in this paper, should be taken into account by the overall framework. Through the dynamic addition and removal of constraints, the knowledge acquired from the environment can be effectively incorporated into the solver, and by exploiting an explanation-providing mechanism and filtering techniques the original plan can be revised in an "intelligent" way.

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 $\mathcal{C}$.

Encoding the domain into a CSP

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. k increases iteratively until a solution is found or a prefixed maximum number is reached. Considering a service domain $\mathcal{SD} = \langle V, Par, Act \rangle$, the target is to encode SD into a $CSP = \langle X_{CSP}, \mathcal{D}, \mathcal{C} \rangle$. First, for each variable $x \in (V \cup Par)$ ranging over D^x for which a state-variable function is defined, and for each $0 \leq i \leq k$ we define a CSP variable $x[i]$ in CSP with domain D^x. The $k\text{-}response$ variables are added to X_{CSP} as they are. Actions are also represented as variables: for each $0 \leq i \leq k-1$ there is a variable $act[i]$ in X_{CSP}, whose domain is the set of all possible action ids in the planning domain, in addition to an idle action $no\text{-}op$: $act[i] \in (Id(Act) \cup$ "noop"). The $no\text{-}op$ action is defined as the action with no preconditions and no effects.

searchForFlight action

prec constraints:

$act[i] =$ "searchForFlight" $\Rightarrow$ $flightWS\_known[i] = false$

effect constraints: /*knowledge-gathering*/

$act[i] =$ "searchForFlight" $\Rightarrow$

$flightWS[i+1] = hotelWS\_response \wedge$

$flightWS\_known[i+1] = true \wedge$

$flightPrice[i+1] = flightPrice\_response \wedge$

$flightPrice\_known[i+1] = true$

bookFlight action

prec constraints:

$act[i] =$ "bookFlight" $\Rightarrow$ $flightWS\_known[i] = true$

effect constraints: /*world-altering*/

$act[i] =$ "bookFlight" $\Rightarrow$

$bookedFlight[i+1] = true \wedge$

$bookedFlight\_changed[i+1] = true$

Figure 2: Constraints encoding the preconditions and effects of the actions offered by the *flightWS*.

Action preconditions and effects, as well as frame axioms, are encoded as constraints on the CSP state variables. So, for example, the constraints corresponding to the operations of the *flightWS* are presented in Figure 2 (where i ranges from 0 to $k-1$). In addition, for every state variable x_{inv} that is not modified by an assignment in $effects(a)$, we add the constraints $act[i] = id(a) \Rightarrow x_{inv}[i] = x_{inv}[i+1]$, $0 \leq i \leq k-1$.

The goal syntax

In the context of loosely-coupled and modular domain descriptions, it is particularly important to support an extended language for expressing user requests in a declarative and accurate syntax, with powerful constructs. The goal language we present accommodates for the specification of expressions on numeric variables, temporal constructs and maintainability properties, adopting a clear distinction between information-gathering and achievement goals.

The goal syntax is defined as follows:

goal	::=	$\wedge_i$(*condition-goal*$_i$ | *subgoal*$_i$)
condition-goal	::=	(*subgoal*) `under_condition` (*condition-goal* | *subgoal*)
subgoal	::=	`achieve-maint` ($\wedge_i$*prop*$_i$) | `achieve`($\wedge_i$*prop*$_i$) | `find_out-maint` ($\wedge_i$*k-prop*$_i$) | `find_out` ($\wedge_i$*k-prop*$_i$)
prop	::=	*var* $\odot$ *value* | *var1* $\odot$ *var2* | (*var1* $\diamond$ *var2*) $\odot$ *value*
k-prop	::=	*k-var* $\odot$ *value* | *k-var1* $\odot$ *k-var2* | (*k-var1* $\diamond$ *k-var2*) $\odot$ *value* | *k-var_known* $=$ *true*

In the above syntax, $var \in Var$, *k-var* stands for the knowledge variables, and *value* represents some boolean or numeric constant, depending on the type of the respective variable. Note that a knowledge proposition *k-prop* refers only to knowledge variables. $\odot$ is a relational operator ($\odot \in \{<, >, \neq, \leq, \geq, =\}$) and $\diamond$ a numeric operator ($\diamond \in \{+, -\}$).

Goal semantics

The `achieve`($\wedge_i prop_i$) subgoal implies that the planner can try any action that has the potential to contribute to the propositions' satisfaction. On the other hand, `find_out`($\wedge_i$*k-prop*$_i$) means that $\wedge_i$*k-prop*$_i$ has to become true at some state, but without employing any actions that include world-altering effects on the variables in $\wedge_i$*k-prop*$_i$. For instance, a goal of the form `find_out`($account\_balance > 100$) will be satisfied if the sensed value for $account\_balance$ is greater than 100, without however allowing any action to alter the variable's value before the sensing action. On the other hand, if the goal is `achieve`($account\_balance > 100$), the planner might try to fulfill it by invoking a pay_in action that increases the $account\_balance$ by some amount. The `maint` annotation adds the requirement that once the respective propositions become true at some state, they should remain true in all subsequent states. In the case of `find_out`, maintainability also enforces that the involved variables should remain intact throughout the whole execution.

Subgoals can be further on combined through the `under_condition` structure: $subgoal_0$ `under_condition` ($subgoal_1$ `under_condition` ($subgoal_2 \ldots$)) is satisfied if each subgoal in the sequence is satisfied at some state, and for each $subgoal_i$, $1 < i \leq n$ that is satisfied at some state, $subgoal_{i+1}$ holds at the directly preceding state. `under_condition` is a powerful operator, and is particularly useful in cases where the user would like to go ahead with altering some variable, only if its sensed value satisfies some property beforehand.

The specification of the action parameters is also part of the goal. It consists of propositions of the form either $par = value$ or $par = k\text{-}var$, where $par \in Par$. The latter is used to express the case where a parameter is not known in advance, but depends on the information acquired through a knowledge variable. The formal semantics of the goal structures can be found in (Kaldeli 2009).

To give an example of a goal, consider a user who wants to reserve a hotel and an air ticket, only if the overall price does not exceed 400 euros, given a travel marketplace that offers the necessary operations. The goal in this case is (Goal 1):

`achieve-maint` (*bookedHotel* $\wedge$ *bookedFlight*)
`under_condition`
(`find_out-maint`($hotelPrice + flightPrice \leq 400$))

The goal is translated into the form of constraints on the CSP state variables, which are added to the constraint set of the *CSP* together with the constraints modelling the domain. Due to lack of space, we do not go through the algorithm for transforming each subgoal and condition goal into constraints, which can be found in (Kaldeli 2009), but rather give an intuition by presenting the constraints encoding Goal 1 in Figure 3.

After the invocation of the solver, an assignment to the action variables is returned, which corresponds to an optimistic plan, i.e. a plan that has the potential to satisfy the goal if the operations respond in the expected way, and the outcome of the sensing actions conforms to the restrictions imposed by the goal.

$bookedHotel[k] \ \wedge \ bookedFlight[k]$

$for\ i \leftarrow\ 0, k-1$ /*maint constraints*/

$\ \ for\ j \leftarrow\ i+1, k$

$\ \ \ bookedHotel[i] \Rightarrow bookedHotel[j]$

$for\ i \leftarrow\ 0, k-1$ /*maint constraints*/

$\ \ for\ j \leftarrow\ i+1, k$

$\ \ \ bookedFlight[i] \Rightarrow bookedFlight[j]$

/*knowledge variables should become known*/

$hotelPrice\_known[k] \wedge carPrice\_known[k]$

$hotelPrice[k] + carPrice[k] \leq 400$

$for\ i \leftarrow\ 0, k-1$ /*maint constraints*/

$\ \ for\ j \leftarrow\ i+1, k$

$\ \ \ hotelPrice[i] + flightPrice[i] \leq 400 \Rightarrow$

$\ \ \ hotelPrice[j] + flightPrice[j] \leq 400$

/*knowledge variables should remain unchanged (find-out goal)*/

$\neg hotelPrice\_changed[k],\ \neg flightPrice\_changed[k]$

$for\ i \leftarrow\ 1, k$ /*under_condition goal*/

$\ (bookedHotel[i] \ \wedge \ bookedFlight[i]) \Rightarrow$

$\ (hotelPrice[i-1] + flightPrice[i-1] \leq 400) \wedge$

$\ (hotelPrice\_known[i-1] \ \wedge \ flightPrice\_known[i-1])$

Figure 3: Constraints encoding Goal 1

Avoiding undesirable situations

The planner performs forward-chaining search, so it may pursue actions that are irrelevant to the goal. Such actions may have undesirable effects, e.g. book also a train ticket, if the respective operation is available, while planning for Goal 1. Therefore, it is important that the returned plan only includes actions that are potentially relevant to the goal and do not have any world-altering effects that the user has not asked for. To this end, a preliminary process that prunes actions irrelevant to the goal is performed. The first phase of this process is done once for the domain, independently of the goal: starting from each action a_i in the domain, we find all actions a_j with at least one effect that has the potential to satisfy one of the preconditions of a_i, so that a Backward Action Chain $BAC(a_i)$ is recursively computed for each action. The second phase involves identifying the actions that have the prospect to satisfy the propositions induced by the goal, and only entail world-altering effects that are either relevant to some proposition of the goal, or are explicitly approved by the user. The details of this process can be found in (Kaldeli 2009). The additional overhead imposed by the second phase of the pruning process is partially counterbalanced by the reduced search space passed to the planner. Figure 4 depicts a broad overview of the framework described so far.

Some initial precaution against contingency during execution should be taken by the domain designer. Considering the goal $\texttt{achieve}(bookedHotel \wedge bookedFlight)$ for example, both plans $\langle$*searchForHotel, bookHotel, searchForFlight, bookFlight*$\rangle$ and $\langle$*searchForHotel, searchForFlight, bookHotel, bookFlight*$\rangle$ would be admissible, if we could guarantee that the execution of all actions succeeds. However, because of the unpredictable nature of WSs, in the first case we might end up paying for a hotel, just to discover later that it is impossible to book a transportations means.

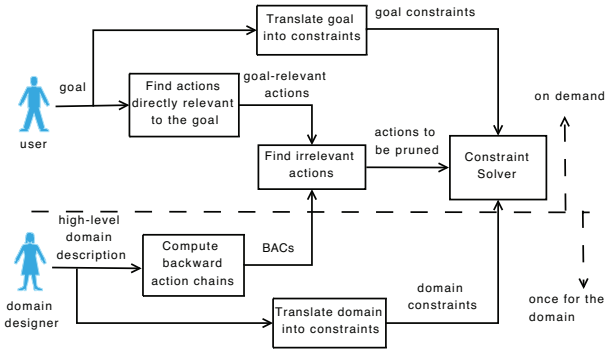

Figure 4: Overview of the planning framework

Such situations are avoided by imposing an action selection order that favours sensing, non-committing actions first.

Concluding remarks

A preliminary implementation of the proposed approach has been developed by using the Choco v2 constraint solving library (www.emn.fr/x-info/choco-solver). We are currently testing the implementation against domains comprising 10-20 actions, and plan to experiment with larger domains. Intended improvements include the application of fine-tuned selection strategies and propagation rules, as well as the investigation of alternative constraint models.

An important extension of our current work involves the support for contingency handling. A framework for interleaving planning and execution would allow for plan revisions based on the feedback gathered at runtime by the executor, triggered either by a newly sensed value or a failure indication.

Acknowledgements

The research is supported by NWO Jacquard **SaS-LeG** contract no. 638.000.000.07N07. Project website: http://www.sas-leg.net.

References

Do, M., and Kambhampati, S. 2000. Solving Planning-Graph by Compiling it into CSP. In *Proc. AIPS-00*, 82–91.

Kaldeli, E. 2009. Using CSP for Adaptable Web Service Composition. Technical Report 2009-7-01, University of Groningen. www.cs.rug.nl/∼eirini/tech_rep_09-7-01.pdf.

Klusch, M., and Gerber, A. 2006. Fast Composition Planning of OWL-S Services and Application. In *ECOWS '06*.

Kuter, U.; Sirin, E.; Nau, D.; Parsia, B.; and Hendler, J. 2004. Information Gathering During Planning for Web Service Composition. In *Journal of Web Semantics*.

Lazovik, A.; Aiello, M.; and Gennari, R. 2005. Encoding Requests to Web Service Compositions as Constraints. In *Proc. CP2005*.

van Beek, P., and Chen, X. 1999. CPlan: A Constraint Programming Approach to Planning. In *Proc. AAAI '99*.