Combining Scalability and Expressivity in the Automatic Composition of Semantic Web Services

Jörg Hoffmann and Ingo Weber
SAP Research, Karlsruhe, Germany
⟨first⟩.⟨last⟩@sap.com

James Scicluna
University of Innsbruck, STI, Austria
james.scicluna@sti2.at

Tomasz Kaczmarek
Poznan University of Economics, Poland
t.kaczmarek@kie.ae.poznan.pl

Anupriya Ankolekar
HP Labs Palo Alto
anupriya.ankolekar@hp.com

Abstract

Automatic web service composition (WSC) is a key component of flexible SOAs. We address WSC at the profile/capability level, where preconditions and effects of services are described in an ontology. In its most expressive formulation, WSC has two sources of complexity: (A) a combinatorial explosion of the services composition space, and (B) worst-case exponential reasoning is needed to determine whether the underlying ontology implies that a particular composition is a solution. Any WSC technology must hence choose a trade-off between scalability and expressivity. We devise new methods for finding better trade-offs. We address (A) by techniques for the automatic generation of heuristic functions. We address (B) by approximate reasoning techniques for the fully expressive case, and by identifying a sub-class where the required reasoning is tractable. We show empirically that our approach scales gracefully to large pools of pre-discovered services, in several test cases.

1 Introduction

In service-oriented architectures, a core operation is web service composition (WSC): combining existing services such that they provide a new desired functionality. Supporting this operation is very important for the flexibility and sustainability of service-oriented architectures. We facilitate WSC for flexible Business Process Management [24] by developing a tool for automatic WSC, integrating the tool within a comprehensive modelling environment and recommending service compositions to the modeler. For this to be successful, the WSC tool must be both sufficiently expressive (for powerful modelling) and sufficiently fast (for good response times).

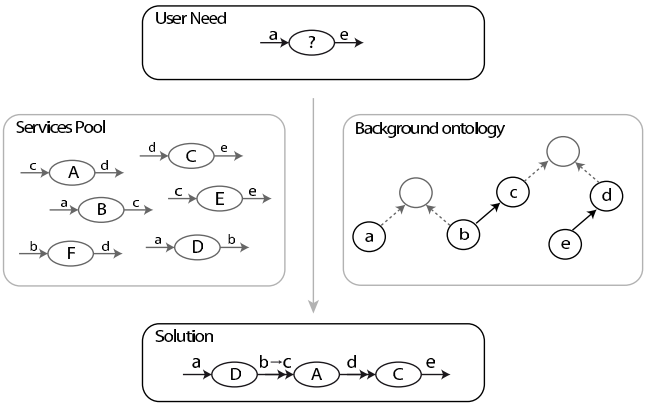

Figure 1. A schematic overview of WSC.

The WSC problem is illustrated in Figure 1. To enable automation, WSC requires web services to be advertised with a description of their functionality, i.e., WSC, in the form we consider here, requires *semantic web services (SWS).* At the so-called profile/capability level, typical SWS approaches such as OWL-S (e.g. [1, 6]) and WSMO (e.g. [8]) describe SWS akin to AI planning formalisms, specifying their input/output parameters, preconditions, and effects. These attributes of SWS are described within an ontology, which formalizes the underlying domain. This kind of WSC has two significant sources of complexity:

- **(A) Combinatorial explosion of possible compositions.** There are exponentially many possible combinations of SWS from the pool. This is particularly challenging since the pool may be large in practice. Note here that we assume that SWS discovery has already taken place, yielding the pool. The outcome of discovery is large, e.g., when many SWS with similar functionalities can be found. Our experiments simulate

this situation.

- **(B) Worst-case exponential reasoning.** To test whether a given combination of SWS is a solution, one must compute the potential outcome of executing the SWS. In the presence of an ontology, this brings about the "frame" and "ramification" problems: the SWS effects may have further implications due to the domain behavior specified in the ontology; it must be determined what those implications are, and whether they affect any of the things that were true before. E.g., if a SWS effect says that an entity c is no longer a member of a concept A, then as an implication c is neither a member of any sub-concept of A anymore. In general, reasoning is needed. In particular, to address the frame problem one needs a notion of minimal change – the outcome state should not differ unnecessarily from the previous state (e.g. a credit card not used by a SWS booking should remain unaffected). Figuring out what is minimal and what is not adds another level of complexity to the required reasoning, and so this reasoning is harder than reasoning in the underlying ontology itself [10].

Given this complexity, it is clearly important to look for good trade-offs between expressivity and scalability. We address (A) by heuristic search (more below). Regarding (B), this problem is closely related to what AI calls "belief update" (e.g. [25, 10]), and we will henceforth refer to the reasoning required for computing the outcome of SWS execution as *update reasoning*. Many existing approaches to WSC (e.g. [21, 17, 20, 2, 23]) simply ignore the ontology, i.e., they act as if no constraints on the domain behavior were given. Most other approaches (e.g. [9, 22, 16, 14]) employ full-scale general reasoners, and suffer from the inevitable performance deficiencies. The middle ground between these two extremes is relatively unexplored but for [7], which devises a method restricting the ontology to be a subsumption hierarchy and [11], which identifies an interesting fragment of description logics (*DL-Lite*) where update reasoning is tractable, but they do not develop an actual WSC tool. In our work, we develop such a tool for a class of ontologies related to DL-Lite; further, we address the fully general case by devising approximate update reasoning techniques.

Our WSC algorithm, Figure 2, uses heuristic search [18], a well known technique for dealing with combinatorial search spaces. The algorithm performs a kind of "forward search", in a space of states s corresponding to different situations during the execution of the various possible compositions. The key ingredients are the *reasoning-startstate*, *reasoning-resultstate*, and *is-solution* procedures, maintaining the search states and detecting solutions; and the *heuristic-function* procedure, taking a state and returning a solution distance estimate h as well as a set

s_0 := *reasoning-startstate*()
(h, H) := *heuristic-function*(s_0)
open-list := $\langle (s_0, h, H) \rangle$;
while TRUE **do**
  (s, h, H) := remove-front(open-list)
  if *is-solution*(s) **then return** path leading to s;
  for all applicable calls a of SWS in H **do**
    s' := *reasoning-resultstate*(s, a)
    (h', H') := *heuristic-function*(s');
    insert-ordered-by-increasing-h(open-list,s',h',H');

Figure 2. The main loop of our WSC algorithm.

H of promising web services. The states are ordered by increasing h (a standard method called "best-first-search" [18]). The set H is used for *filtering* the explored SWS calls. We will see that both h and H, and especially their combination, bring huge scalability gains.

We base our work on a natural formalism for WSC, with a semantics following the *possible models approach* (PMA) [25]. The PMA addresses the frame and ramification problems via a widely adopted notion of minimal change; it underlies all recent work on formal semantics for WSC (e.g. [15, 4, 11]). Our technical contributions are:

- We develop the *heuristic-function* procedure above, by suitably adapting AI Planning techniques [12].[1] This has been attempted for WSC only in a single research effort [16] as yet. Going beyond that work, and beyond all related work in planning, ours is the first technology that takes domain knowledge, as given in the ontology, into account in the heuristic.

- We show that, if the ontology specifies binary clauses only, then update reasoning is tractable. Note that this is not self-evident from the fact that reasoning over binary clauses is tractable: e.g. it is known [10] that, for Horn clauses, update reasoning is *not* tractable. Binary clauses are related to, but not a subset or superset of, DL-Lite (details in Section 6).

- We open up a new way to more fine grained trade-offs between expressivity and scalability: *approximate update reasoning*. Instead of restricting the ontology up to a point where update reasoning is tractable, we propose to approximate the update reasoning itself. We

[1] For the reader not familiar with the field of AI Planning, we remark that this kind of heuristic function, used in a forward search as per Figure 2, is since almost a decade by far the most successful method in planning. In particular, in almost all known benchmark domains it is far more efficient than, e.g., approaches based on Graphplan [5] or partial-order planning [19].

identify certain properties of updates that can be exploited to design techniques for under-approximation and over-approximation. We show how either of soundness or completeness can be guaranteed by interleaving both approximations. Empirical exploration of these techniques is beyond the scope of this paper, but forms a promising topic for future research.

- Web service outputs are naturally modelled as new ontology instances: e.g., if a service makes a flight reservation, then this is modelled as a new instance of the respective concept. Previous WSC tools (e.g. [16]) have taken the same approach, but were limited to consider pre-fixed sets of "potential" outputs. By contrast, our algorithms keep track of on-the-fly creation of outputs.

We implemented our algorithms, for the case of binary clauses, in a WSC tool. We run experiments on two test cases one of which stems from a case study in the telecommunications sector. The results show how our heuristic and filtering techniques enable us to scale to large SWS pools.

Sections 2, 3, and 4 respectively introduce our WSC formalism, update reasoning algorithms, and heuristic function. Empirical results are in Section 5. Related work is discussed in Section 6, and Section 7 concludes. To improve accessibility, many technicalities – including in particular all proofs as well as the details regarding approximate update reasoning – are moved into a longer TR [13].

2 Formalizing WSC

We introduce a formalism for WSC, denoted $\mathcal{WSC}$. The input/output behavior of SWS maps to input/output parameters, on which preconditions and effects are specified. This closely follows the specification of SWS at the OWL-S "service profile" level and at the WSMO "service capability" level. We first introduce the syntax, then the semantics. For space reasons, we limit both to what is necessary to understand our contribution.

We use standard terminology from logics, with predicates G, H, I, variables x, y, and constants c, d, e (ontology "instances"); equality is a "built-in" predicate. *Literals* are possibly negated predicates whose arguments are variables or constants; if all arguments are constants, the literal is *ground*. Given a set X of variables, we denote by $\mathcal{L}^X$ the set of all literals which use only variables from X. If l is a literal, we write $l[X]$ to indicate that l uses variables X. If $X = \{x_1, \ldots, x_k\}$ and $C = \{c_1, \ldots, c_k\}$, then by $l[c_1, \ldots, c_k/x_1, \ldots, x_k]$ we denote the substitution, abbreviated $l[C]$. In the same way, we use the substitution notation for any construct involving variables. By $\bar{l}$, we denote the inverse of literal l. If L is a set of literals, then $\overline{L}$ denotes $\{\bar{l} \mid l \in L\}$, and $\bigwedge L$ denotes $\bigwedge_{l \in L} l$.

An *ontology* Ω is a pair $(\mathcal{P}, \Phi)$ where $\mathcal{P}$ is a set of predicates and Φ is a conjunction of closed first-order formulas. We call Φ a *theory*. A *clause* is a disjunction of literals with universal quantification on the outside, e.g. $\forall x. \neg G(x) \vee H(x) \vee I(x)$. A clause is *Horn* if it contains at most one positive literal. A clause is *binary* if it contains at most two literals. Φ is Horn/binary if it is a conjunction of Horn/binary clauses. Note that binary clauses can be used to specify many common ontology properties such as subsumption relations $\forall x. G(x) \Rightarrow H(x)$ ($\phi \Rightarrow \psi$ abbreviates $\neg\phi \vee \psi$), attribute image type restrictions $\forall x, y. G(x, y) \Rightarrow H(y)$, and role symmetry $\forall x, y. G(x, y) \Rightarrow G(y, x)$. An example of a property that is Horn (but not binary) is role transitivity, $\forall x, y, z. G(x, y) \wedge G(y, z) \Rightarrow G(x, z)$.

A *web service* w is a tuple $(X_w, \text{pre}_w, Y_w, \text{eff}_w)$, where X_w, Y_w are sets of variables, pre_w is a conjunction of literals from $\mathcal{L}^{X_w}$, and eff_w is a conjunction of literals from $\mathcal{L}^{X_w \cup Y_w}$. I.e., X_w are the inputs and Y_w the outputs, i.e., the new constants created by the web service; pre_w is the precondition, eff_w the effect (sometimes referred to as the *postcondition* in the literature). Before a web service can be applied, its inputs and outputs must be instantiated with constants, yielding a *service*; to avoid confusion with the search states s, we refer to services as (web service) *applications*, denoted with a. Formally, for a web service $(X, \text{pre}, Y, \text{eff})$ and tuples of constants C_a and E_a, a service a is given by $(\text{pre}_a, \text{eff}_a) = (\text{pre}, \text{eff})[C_a/X, E_a/Y]$. The web service's inputs are instantiated with C_a, and its outputs are instantiated with E_a.

Tasks are tuples $(\Omega, \mathcal{W}, \mathcal{U})$, i.e., the collection of all aspects relevant to a specific composition request. Ω is an ontology and $\mathcal{W}$ is a set of web services. $\mathcal{U}$ is the user requirement, a pair $(\text{pre}_\mathcal{U}, \text{eff}_\mathcal{U})$ of precondition and effect. By $C_\mathcal{U}$, we will denote a set of constants corresponding to the variables of the user requirement precondition. These will be the only constants taken to exist initially. In other words, these variables become "generic constants" about which we know only the user requirement precondition; we want to apply web services from $\mathcal{W}$ to reach a situation where an appropriate instantiation of the variables in $\text{eff}_\mathcal{U}$ is guaranteed to exist.

The semantics of our formalism relies on a notion of *beliefs*, where each belief is a set of *models*. A "model" corresponds to a particular situation during the execution of a composition. Now, since the precondition of the user requirement does not completely describe every aspect of the status of the ontology instances, several models are possible initially. Similarly, since web service effects do not completely describe every aspect of the changes they imply, executing a web service can result in several possible models. This uncertainty about the actual situation is formalized in terms of beliefs, containing the entire set of models possible at a given point in time.

Models m are pairs (C_m, I_m) where C_m is the set of constants that exist, and I_m is the interpretation of the predicates over these constants. By $m \models \phi$ for a closed first-order formula ϕ, we mean that $I_m \models \phi$ where the quantifiers in ϕ range over C_m. The *initial belief* b_0 is undefined if $\Phi \wedge \text{pre}_{\mathcal{U}}$ is not satisfiable; else, $b_0 := \{m \mid C_m = C_{\mathcal{U}}, m \models \Phi \wedge \text{pre}_{\mathcal{U}}\}$. A *solved belief* is a belief b s.t. there exists a tuple C of constants s.t., for all $m \in b$, $m \models \text{eff}_{\mathcal{U}}[C]$. The core definition states how services affect models; we adopt this definition from the widely used PMA semantics [25]. Assume a model m and a service a. The *result* of applying a to w is $res(m, a) :=$

$$\{(C', I') \mid C' = C_m \cup E_a, I' \in min(m, C', \Phi \wedge \text{eff}_a)\}$$

Here, $min(m, C', \phi)$ is the set of all C'-interpretations that satisfy ϕ and that are minimal with respect to the partial order defined by $I_1 \leq I_2$:iff for all propositions p over C_m, if $I_2(p) = I_m(p)$ then $I_1(p) = I_m(p)$. In words, a C'-interpretation is in $min(m, C', \phi)$ iff it satisfies ϕ, and is as close to m as possible. To illustrate this definition, consider the following informal example.

Example 1 *Let p, q, r be propositions, $\Phi = \neg p \vee \neg q \vee \neg r$. Let m be a model where $p = 1, q = 1, r = 0$. Let a be a service with precondition $p \wedge q$ and effect r. What happens if we apply a to m?*

If we simply set $r = 1$, the model we get is $p = 1, q = 1, r = 1$. This model is inconsistent with Φ. The PMA resolves this inconsistency by making changes to what we inherited from m. Namely, p and q are not mentioned by the effect; their values are inherited. Setting one of them to 0 resolves the conflict; if we set both to 0, then the change to m is not minimal. Hence we get two resulting models: $p = 0, q = 1, r = 1$ and $p = 1, q = 0, r = 1$.

As stated, the PMA is used in all recent works on formal semantics for WSC (e.g. [15, 4, 11]). Alternative belief update semantics from the AI literature could be used in principle; this is a topic for future work.

We say that a is *applicable* to m if $C_a \subseteq C_m$, $m \models \text{pre}_a$, and $res(m, a) \neq \emptyset$: the inputs exist, the precondition holds, and the result is not empty (the result is empty in case of unresolvable conflicts between eff_a and Φ). Assume a belief b. If a is applicable to all $m \in b$, then $res(b, a) := \bigcup_{m \in b} res(m, a)$; else, $res(b, a)$ is undefined.[2] The res function is extended to sequences $\langle a_1, \ldots, a_n \rangle$ in the obvious way. A *solution for* b is a sequence $\vec{a}$ s.t. $res(b, \vec{a})$ is a solved belief; a solution for b_0 is a solution for the task.

[2]Requiring that a is applicable to *all* $m \in b$ corresponds to what is usually called *plug-in matches*: the ontology implies that a is always applicable. A more general notion are *partial matches*, where a is applicable to *at least one* $m \in b$. We do not consider partial matches because, for them, update reasoning is coNP-complete even with empty Φ.

3 Update Reasoning

We now specify what the search states s from Figure 2 (c.f. Section 1) are, and how they are maintained. We first explain the basic aspects of our search; then we discuss the difficulties with unrestricted Φ, and with Horn Φ; then we show that binary Φ can be dealt with efficiently; then we summarize our approximations for the general case.

Our approach is based on *forward search*. We search for a solution in the space of beliefs b that can be reached by chaining services starting from b_0. The elementary steps in such a search are: testing whether a belief b is a solution; testing whether a service a is applicable to a belief b; and computing the outcome $res(b, a)$ of such a service. The question arises: How do we represent beliefs? And even: Which *aspects* of the beliefs do we represent?

Herein, we focus on methods that maintain only a partial knowledge about the beliefs b. The obvious advantage is that we avoid the prize for exact representation of beliefs (each of which may contain exponentially many models); the disadvantage is that such methods are not always applicable, or may sacrifice some precision. Note that maintaining the existing constants is easy. Since every model $m \in b_0$ has $C_m = C_{\mathcal{U}}$, and since every service adds the same new constants to each model, we have that $C_m = C_{m'}$ for every $m, m' \in b$ for every reachable b. We will hence ignore this issue for the remainder of this section, and concentrate fully on the interpretations, I_m.

The minimum knowledge we need to maintain about each b is *the set of literals that are true in all models of b*, $\bigcap_{m \in b}\{l \mid I_m \models l\} =: Lits(b)$. Based on $Lits(b)$, we can determine whether a is applicable, namely iff $\text{pre}_a \subseteq Lits(b)$, and whether b is solved, namely iff there exists C s.t. $\text{eff}_{\mathcal{U}}[C] \subseteq Lits(b)$. So we define the search states s from Figure 2 as pairs (C_s, L_s); if b is the belief reached via the same service sequence, then we want to have $C_s = C_m$ for $m \in b$, and $L_s = Lits(b)$.

How do we maintain those s? There are two bad news. The first of those is that computing $Lits(res(b, a))$ is hard, even if Φ is Horn; this can be proved based on earlier work in the area of belief update [10]:[3]

Theorem 1 *Assume a $\mathcal{WSC}$ task $(\Omega, \mathcal{W}, \mathcal{U})$ with fixed arity. Assume a model m, a service a, and a literal l such that $m \models l$. It is Π_2^p-complete to decide whether $l \in Lits(res(m, a))$. If Φ is Horn, then the same decision is coNP-complete.*

So, even if we are considering only a single model m, it requires exponential effort to determine $Lits(res(m, a))$.

[3]By *fixed arity*, we mean a constant upper bound on predicate arity, on the number of input/output parameters of any web service, and on the depth of quantifier nesting within Φ. This is a reasonable restriction in practice; e.g., predicate arity is at most 2 in DL.

Note here that each decision problem (general/Horn) has worse complexity than reasoning in the respective fragment of logics. Further:

Theorem 2 *There exist a $\mathcal{WSC}$ task $(\Omega, \mathcal{W}, \mathcal{U})$ where Φ is Horn, a service a, and two beliefs b and b' so that $Lits(b) = Lits(b')$, but $Lits(res(b, a)) \neq Lits(res(b', a))$.*

This is because it can happen that $b \neq \{m \mid m \models \Phi \wedge Lits(b)\}$. So, even if we had an oracle computing belief updates, $L_s = Lits(b)$ would not always provide enough information for that computation.

The good news is: if we consider binary clauses instead of Horn clauses, then the difficulties disappear. This can be shown through a series of technical observations. The core observation characterizes the situation where a literal $l \in Lits(b)$ "disappears":

Lemma 1 *Assume a $\mathcal{WSC}$ task $(\Omega, \mathcal{W}, \mathcal{U})$. Assume a belief b, a service a, and a literal $l \in Lits(b)$. Then, $l \notin Lits(res(b, a))$ iff there exists a set L_0 of literals satisfied by a model $m \in b$, such that $\Phi \wedge \mathit{eff}_a \wedge \bigwedge L_0$ is satisfiable and $\Phi \wedge \mathit{eff}_a \wedge \bigwedge L_0 \wedge l$ is unsatisfiable.*

From this lemma, it is not difficult to conclude that, with binary Φ, a literal disappears only if its opposite is necessarily true, i.e., if $l \in Lits(b)$ but $l \notin Lits(res(b, a))$, then $\Phi \wedge \text{eff}_a \models \bar{l}$. A relatively easy observation is that, for any Φ, if $l \notin Lits(b)$ but $l \in Lits(res(b, a))$, then $\Phi \wedge \text{eff}_a \models l$. We get:

Theorem 3 *Assume a $\mathcal{WSC}$ task $(\Omega, \mathcal{W}, \mathcal{U})$ with fixed arity, where Φ is binary. Assume a belief b, and a service a; let $L := \{l \mid \Phi \wedge \mathit{eff}_a \models l\}$. Then $Lits(res(b, a)) = L \cup (Lits(b) \setminus \overline{L})$. Given $Lits(b)$, this can be computed in time polynomial in the size of $(\Omega, \mathcal{W}, \mathcal{U})$.*

Theorem 3 corresponds directly to a way of dealing with the states s from Figure 2: each s is a pair (C_s, L_s) as explained above; given a, we test whether $\text{pre}_a \subseteq L_s$; if so, we compute (in polynomial time) $L := \{l \mid \Phi \wedge \text{eff}_a \models l\}$, and take the successor state to be $(C_s \cup E_a, L \cup (Lits(b) \setminus \overline{L}))$ where E_a denotes a's output constants; we stop when we found s so that ex. $C \subseteq C_s$ s.t. $\text{eff}_{\mathcal{U}}[C] \subseteq L_s$.

Of course, in practice it is necessary to deal with more general ontologies, e.g. featuring arbitrary clauses. One option is to sacrifice some precision by mapping Φ into a more tractable case, e.g. projecting each clause into a binary clause. Such an approach, however, does not preserve soundness – the returned solution may make use of constraints that do not hold – and neither does it preserve completeness – the task may only be solvable under weaker constraints.

It turns out one can preserve either of soundness or completeness by approximating dynamically instead of statically: *instead of approximating the input to the composition, approximate the update reasoning that it performs.* Since the details of this approach are somewhat intricate, they are removed from this paper, for the sake of accessibility; they are available in [13]. In a nutshell, the approach works as follows. The algorithm now maintains search states $s = (C_s, L_s^-, L_s^+)$ – in difference to the pairs (C_s, L_s) that we used before. Our approximate update algorithm ensures that L_s^- under-approximates $Lits(b)$, i.e., $L_s^- \subseteq Lits(b)$, and that L_s^+ over-approximates $Lits(b)$, i.e., $Lits(b) \subseteq L_s^+$. By virtue of these properties, if one tests solutions and service applicability based on L_s^-, then the WSC tool is sound but incomplete; if one tests solutions and service applicability based on L_s^+, then the WSC tool is complete but not sound. The approximate update guarantees the approximation properties by using L_s^- and L_s^+ as the basis for approximating either of the two sides of the equivalence proved in Lemma 1.

4 Heuristic Function

We now present techniques to effectively navigate the forward search. We develop a heuristic function ("h" in Figure 2), as well as a filtering technique ("H" in Figure 2). The heuristic function is inspired by successful techniques from AI Planning [12]. The technical basis is an approximate WSC process, where the main approximation is to act as if both a literal and its negation could be true at the same time.[4] Given a state s, the approximate WSC finds an approximate solution for s; h and H are then easily obtained from the approximate solution. We will see that the approximate WSC is computationally cheap enough to make its solution in every s feasible. Going beyond all previous approximations of its kind, we show how the approximate WSC can deal with ontologies, and with on-the-fly creation of new constants. We consider the general case, arbitrary Φ; our techniques for binary Φ are obtained from that in the obvious way.

procedure *heuristic-function*(s)
(1) (t_{max}, C, L, A, S) := *build-ACG*(s)
(2) **if** $t_{max} = \infty$ **then return** $(\infty, \emptyset)$
(3) $\langle A_0, \ldots, A_{t-1} \rangle$:= *extract*(t_{max}, C, L, A, S)
(4) **return** $(\sum_{i=0}^{t_{max}-1} |A_i|, \{w \in \mathcal{W} \mid a \in A_0 \text{ is based on } w\})$

Figure 3. Heuristic function main control.

The approximate WSC consists of a forward step, *build-ACG*, and of a backward step, *extract*. Figure 3 shows how

[4] Note that this approximation is much more severe than approximate update reasoning as discussed in Section 3.

these are arranged. First, the forward step returns a tuple (t, C, L, A, S). The meaning of this tuple will become clear below; t is an estimation of how many steps it takes to achieve $\text{eff}_{\mathcal{U}}$, starting from s. If $t = \infty$ then we know that $\text{eff}_{\mathcal{U}}$ cannot be reached from s, and we can prune s from the search. Otherwise, the backward step, *extract*, is called, returning an approximate solution $\langle A_0, \ldots, A_{t-1} \rangle$ where each A_i is a set of services. The heuristic function returns the count of services as h, and the set of web services participating in A_0 as H.

procedure *build-ACG*(s)
/* s must contain information providing C and L */
(1) $t := 0$; $C_0 := C$ s.t. $C \supseteq C_b$; $L_0 := L$ s.t. $L \supseteq Lits(b)$
(2) **while** not ex. $C \subseteq C_t$ s.t. $\text{eff}_{\mathcal{U}}[C] \subseteq L_t$ **do**
(3) create $\{e^1_{t+1}, \ldots, e^M_{t+1}\}$ s.t. $\{e^1_{t+1}, \ldots, e^M_{t+1}\} \cap C_t = \emptyset$
(4) $C_{t+1} := C_t \cup \{e^1_{t+1}, \ldots, e^M_{t+1}\}$; $L_{t+1} := L_t$
(5) $A_t := \{w[C/X_w, (e^1_{t+1}, \ldots, e^{|Y_w|}_{t+1})/Y_w] \mid w \in \mathcal{W}, C \subseteq C_t, \text{pre}_w[C] \subseteq L_t\}$
(6) **for** all $a \in A_t$ **do**
(7) **for** all $l \notin L_{t+1}$ s.t. $\Phi \wedge \text{eff}_a \models l$ **do**
(8) $L_{t+1} := L_{t+1} \cup \{l\}$, $S(l) := a$
(9) **if** $\sigma_e(L_{t-K+1}) = \cdots = \sigma_e(L_{t+1})$ **then return** ∞
(10) $t := t + 1$
(11) $t_{max} := t$
(12) **return** (t_{max}, C, L, A, S)

Figure 4. Building an ACG.

Figure 4 shows pseudo-code for *build-ACG*. The notations are as follows. C_b is the set of constants in the belief (recall that all models within a belief share the same constants, c.f. Section 3). M is the maximum number of outputs any web service has. K is the maximum over: 1; the number of input variables of any web service whose precondition contains inequality; and the number of variables in $\text{eff}_{\mathcal{U}}$ if that contains inequality. The σ_e function maps any e^i_t generated in any layer t to a constant e^i.

The algorithm builds a structure that we refer to as the *approximated composition graph* (ACG) for s. The ACG is a tuple (t_{max}, C, L, A, S) where:

- t_{max} is the number of *time steps* in the ACG. At each time step t between 0 and t_{max}, the ACG represents an approximation of what can be achieved within t steps. t_{max} is the index of the highest time step built by the algorithm. This either means that the ACG reached $\text{eff}_{\mathcal{U}}$ (the user requirement effect) for the first time at t_{max}; or that the ACG reached a fixpoint without reaching $\text{eff}_{\mathcal{U}}$, in which case $t_{max} = \infty$.

- C is a vector of sets of constants, indexed by time steps t. Each C_t contains the set of constants that is considered reachable by the ACG, within t steps.

- L is a vector of sets of literals, indexed by time steps t. Each L_t contains the set of literals that is considered reachable by the ACG, within t steps. If $l \in L_t$, then the interpretation is that l can become known (true in all models of a belief) within t steps. *L_t may contain both a literal and its negation; both are considered reached, i.e., there is no handling of conflicts.*

- A is a vector of sets of services, indexed by time steps t. Each A_t is the set of services that is considered reachable by the ACG, within t steps.

- S is a function from literals to services. The meaning of $S(l) = a$ is that, if t is the first time step where l is reached in the ACG, then a can be used at time $t-1$ to achieve l at time t. S will be used to extract an approximate solution.

Line (1) of Figure 4 initializes the ACG, in an obvious way. Note that s is required to provide supersets of C_b and $Lits(b)$. With our techniques from Section 3, this will be C_s and L_s for the case of binary clauses, where we have $C_s = C_b$ and $L_s = Lits(b)$; it will be C_s and L^+_s for the general case and approximate update reasoning, where we have $C_s = C_b$ and $L^+_s \supseteq Lits(b)$. The ACG algorithm as specified works with any representation of search states, as long as appropriate $C \supseteq C_b$ and $L \supseteq Lits(b)$ can efficiently be extracted from it.

Lines (2) to (10) loop until the goal is reached in L_t, or until line (9) has stopped the algorithm. Each loop iteration extends the ACG by another time step, $t + 1$. Lines (3) to (5) set the constants at $t + 1$, and the services at t. This is straightforward.; the only subtlety is the creation of new constants, which we discuss below. Lines (6) to (8) include all new literals that can be deduced from the effect of a service in A_t; the S function is set accordingly.

Line (9) is a fixpoint test. The test is non-trivial in that it takes the creation of new constants into account: since the ACG creates a set of new constants at every time step, it never reaches a fixpoint in the naive sense. However, we can notice that all new constants behave exactly like the old constants. If that is the case, then extending the ACG further will not get us anywhere, unless some web service (or the goal) requires several different constants with the same properties. The latter is captured by K as explained above; using the mapping σ_e, line (9) tests whether no relevant progress has been made in the last $K + 1$ steps.

A remarkable trick used in *build-ACG* is its handling of constants creation. We generate a fixed number of new constants – the maximum number of outputs of any web service – per ACG layer, and let all services output *the same* constants. This saves us from exponential growth! If one allows different output constants for each service, then the number of constants may grow by a multiplicative factor in

each time step, and the ACG size is exponential in t. A simple example: there are two web services w_{GH} and w_{HG}, the former with input x of concept G and output y of concept H, the latter vice versa. There are 2 constants initially, one in G one in H. At $t = 0$ we get one service for each web service, and hence at $t = 1$ we get 2 new constants. At $t = 1$ we get 2 services for each web service, at $t = 2$ we get 4 new constants, etc. In this example the exponential growth is obviously redundant, but that redundancy is far from obvious in the general case.

Our ACG algorithm has all the desirable properties of an over-approximation underlying a heuristic function:

Theorem 4 *Assume a $\mathcal{WSC}$ task $(\Omega, \mathcal{W}, \mathcal{U})$ with fixed arity. Assume a belief b and corresponding search state s. Let n be the length of a shortest solution for b, or $n = \infty$ if there is no such solution. Then* build-ACG(s) *returns t_{max} so that $t_{max} \leq n$; if* build-ACG(s) *uses an oracle for SAT, then it terminates in time polynomial in the size of $(\Omega, \mathcal{W}, \mathcal{U})$ and s.*

Of course, we do not have an "oracle for SAT". But if Φ falls into a tractable fragment, such as Horn or binary Φ, then the algorithm takes polynomial time.

Theorem 4 follows from three technical observations: (A) If $\langle a_0, \ldots, a_{n-1} \rangle$ is a solution for b, then *build-ACG*(s) run without line (9) returns t_{max} so that $t_{max} \leq n$. (B) If the condition tested in line (9) holds in iteration t, then, continuing the algorithm, it will hold for all $t' > t$. (C) The number of time steps of the ACG is bound by an expression exponential only in predicate arity.

By Theorem 4, we know that t_{max} is a lower bound on solution distance. This is nice because algorithms such as A* can use it to find provably shortest solutions [18]. However, we found that, using t_{max} as a heuristic function, A* fails to solve any but the tiniest examples. We address this problem by devising a different heuristic function: an additional *backward step* is performed on the ACG, to extract an approximated solution, which delivers much better search guidance in practice. The algorithm is straightforward, and we move it to [13] in order to spare the reader some details. In a nutshell, the approximate solution extraction chains backwards from the goal, using the S function to select supporting actions for sub-goal literals, and inserting action preconditions as new sub-goals. The selected actions form the returned solution $\langle A_0, \ldots, A_{t_{max}-1} \rangle$. We remark that, in difference to t_{max}, $\sum_{i=0}^{t_{max}-1} |A_i|$ is *not* a lower bound on solution length: the function S commits to a particular choice how to support a sub-goal, although different combinations of such choices may result in different solutions. However, our empirical results confirm that $h := \sum_{i=0}^{t_{max}-1} |A_i|$ and $H := \{w \in \mathcal{W} \mid a \in A_0 \text{ is based on } w\}$) (c.f. Figure 3 deliver very useful search guidance in practice.

5 Empirical Results

We implemented a tool handling binary clauses as per Theorem 3, and computing heuristic (h) and filtering (H) information as per Section 4. The implementation language is Java. The tool accepts a set of SWS and user requirement descriptions in the WSMO formalism, specified in a subset of the WSML language. Our experiments were run on a laptop with a Pentium M CPU running at 2.0 GHz, reserving up to 1GB of main memory for the tool. To assess the benefits of both heuristic techniques – h and H – we experimented with all possible configurations: **Blind** uses neither h nor H; **Heuristic** uses only h; **Filtering** uses only H; **Full** uses both.

We created testbeds simulating the expected surrounding conditions of WSC in practice: we keep the size of the solutions moderate; the parameter we scale is *the size of the SWS pool*. This corresponds to the common intuition about WSC, that solutions tend to be simple but finding the right services to incorporate is difficult. Note that, as stated, **the SWS pool corresponds to the outcome of discovery.** Hence our scaling scenario addresses the case where many SWS can be found and, without performing the actual composition, it is not possible to filter out those few SWS that are actually needed. When discovering in a large environment, one would expect that many alternative implementations of each required functionality can be found; the alternative implementations are similar but not identical – like different SWS offering flight connections. We simulate this situation as follows. In addition to the SWS that are needed for the solution, we add N additional "randomized" SWS into the pool. All randomized SWS use (only) the original ontology, and we generate them by randomly modifying the original SWS. Say max is the maximum number of literals appearing in the precondition/effect of any original SWS. Then each of the N randomized SWS is generated by, uniformly: choosing one of the original SWS; choosing numbers $0 \leq k, l \leq max$; choosing k literals to add to the precondition; and choosing l literals to add to the effect.

We designed two test cases, called TPSA and VTA. The latter is a variant of the well-known virtual travel agency, where transport and accommodation etc. need to be booked based on a trip request. TPSA comes from a use case in the telecommunications sector[5] and describes a scenario in which a client requests a Voice over IP (VOIP) service. The WSC task is to automatically compose a process setting up the VOIP in the TP system. This process involves identifying the required hardware, setting up the contract, saving the contract within the CRM system, etc. The user requirement is to obtain an invoice confirming the activation of the

[5]This use case stems from Telekomunikacja Polska (TP) and is a part of the SUPER Integrated Project (under the EU's FP6), in the context of Semantic Business Process Management.

VOIP. In both VTA and TPSA, the shortest solution contains 7 web services.

Figure 5 shows our results, plotting runtime for the four configurations over N, for one instance per each setting of N. Note that the scale for TPSA is logarithmic to improve readability. We applied a runtime cut-off of 10000 seconds. (**Blind** actually ran overnight on TPSA with 20 randomized SWS, without finding a solution). Note that the runtime sometimes does not grow monotonically over N; this is just due to the randomization, and would disappear when taking mean values over several runs.

The data clearly show that the heuristic techniques bring a vast advantage over the blind search. We can also see that the importance of the different techniques, solution distance estimation, h, or filtering, H, depends on the domain. In TPSA, if H is used, we get linear runtime behavior and the effect of h is only cosmetic. In VTA, on the other hand, using only h (**Heuristic**) is much better than using only H (**Filtering**). We also see in VTA that there can be synergy in the combination of the two techniques: **Full** works vastly better than any other configuration. In fact, this configuration is more effective than **Filtering** in the this scenario (VTA). This is due to the more parallel nature of the solution of the VTA setting, in contrast to the more sequential TPSA solution, where the difference between **Filtering** and **Full** is not so significant. I.e., there are several SWS in VTA which could be executed in parallel, resulting in more actions which are still to be considered after **Filtering**.

Due to the artificial nature of our randomized SWS, the observed advantage of the heuristic techniques over blind search might be more extreme than what one would get with real large SWS pools (which do not yet exist). However, one can reasonably expect that the overall patterns of behavior will be similar.

A sensible comparison to alternate WSC tools is difficult due to those tools' widely disparate nature (beside being technically challenging due to widely disparate input languages). We ran tests with the DLVK tool [9], which is based on general reasoning (answer set programming). We chose DLVK because it is publicly available, and its language is expressive enough to handle our test cases. In fact, DLVK allows more general Φ than our tool; so the question answered by the experiment is whether our tool actually gains something, by giving up some expressivity. It turns out that DLVK is much slower even than **Blind**. With $N = 0$, DLVK takes 12 minutes for TPSA and 2 hours for VTA. In both test cases, DLVK runs out of time for $N \geq 5$. These results should not be over-interpreted, since a direct comparison between DLVK and our tool is unfair. But the results certainly show that the trade-off between expressivity and scalability is important.

6 Related Work

One large difference of our work to almost all other approaches to WSC is that we devise a heuristic function guiding the search. There exists only a single other work, namely [16], that adapts such a technique for WSC.[6] [16] is related to our work in that, like us, it is inspired by [12]. However: [16] do not take the ontology constraints Φ into account in the heuristic function; they use a worst-case exponential reasoner to determine their search states, and they explicitly enumerate the models in every belief; they restrict the creation of new constants to a single one per ontology concept; and they show results only for a single small example task.

Another major difference of our work to almost all related works lies in our exploration of the trade-off between expressivity of Φ, and the required reasoning about the consequences of applying web services. Many approaches act as if Φ is empty (e.g. [21, 17, 20, 2]), and most other approaches employ full-scale general reasoners (e.g. [9, 22, 16, 14]). To the best of our knowledge, the only exceptions are [7] and [11].

In [7], the background ontology is a subsumption hierarchy. This is compiled into intervals, where each interval represents a concept and the contents are arranged to correspond to the hierarchy. The intervals are used for matching the web services during composition, where a notion of "switches" is used to be able to construct solutions dealing with partial matches. Search proceeds in a depth-first fashion, with no heuristic information. Hence, by comparison to our work, [7] uses a more general notion of matches, but a more restrictive notion of ontologies, and lacks our techniques for search guidance. A combination of both techniques might be interesting to explore.

[11] investigates, in DL-Lite, what happens if instance data is updated at the DL ABox level, and the updated belief shall be represented as a new ABox. DL-Lite is more powerful than our binary clauses in some aspects, and less powerful in other aspects. All clauses (Φ statements) in DL-Lite are binary. However, [11] allow unqualified existential quantification, membership assertions (ABox literals) using variables, and updates involving general (constructed) DL concepts. On the other hand, DL-Lite does not allow clauses with two positive literals, DL-Lite TBoxes allow literals on roles only if one of the 2 variables is existentially quantified, and DL-Lite (like any DL) does not allow predicates of arity greater than 2. Also, in difference to us, [11] do not develop an actual WSC tool. Our heuristic techniques are certainly compatible with (a subset of) DL-Lite updates, and so an exciting topic remains to combine the two, yielding scalable WSC technology for an interesting

[6] [14] follow a more limited approach, by compilation into actual planning formalisms.

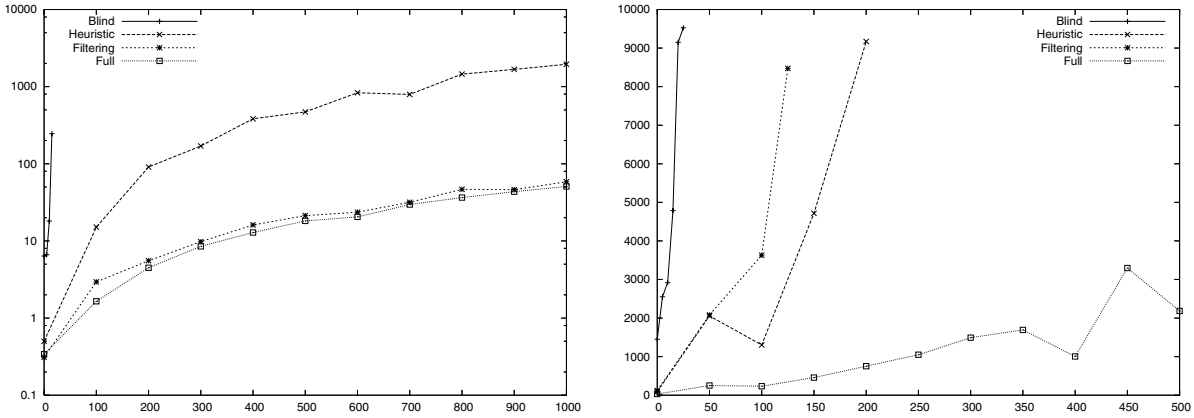

Figure 5. Results for TPSA (left) and VTA (right). Runtime (y-axis, seconds) plotted over N (x-axis), i.e., the number of randomized SWS. Note the logarithmic runtime scale for TPSA.

fragment of DL.

There exist other static compositional approaches which usually imply encoding a composed solution using some appropriate language. An example of such a language is BPEL [3]. Needless to say, such static descriptions are not adequate for automated composition as we envision in this paper. We do believe however that solutions composed using semantic technologies can be linked to such static languages. A SWS is generally linked to an existing service such as WSDL. This link is referred to as *grounding*. While our tool does not deal with grounding so far, the translation of a found composition solution to an executable language such as BPEL should be straight-forward – given the groundings of the utilized SWS are specified. However, the details may be tricky, and the success of an automatic generation of an executable process from a composition cannot be guaranteed in general. Thus, testing and other means of quality assurance are highly advisable in productive settings.

7 Conclusion

Automatic WSC is a core feature of flexible service-oriented architectures. Our implemented tool provides a uniquely strong trade-off between expressivity and scalability, in that it allows non-trivial ontologies without resorting to worst-case exponential reasoning, and in that it successfully employs heuristic and filtering techniques.

In the near future, we will design methods tackling Business Policies, and optimizing the QoS performance of the composed service.

References

[1] A. Ankolekar et al. DAML-S: Web Service Description for the Semantic Web. In *International Semantic Web Conference*, 2002.

[2] V. Agarwal, G. Chafle, K. Dasgupta, N. Karnik, A. Kumar, S. Mittal, and B. Srivastava. Synthy: A System for End to End Composition of Web Services. *Journal of Web Semantics*, 3(4), 2005.

[3] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guzar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. *Web Services Business Process Execution Language*. OASIS, version 2.0 edition, 23rd Aug. 2006. Public Review Draft, http://docs.oasis-open.org/wsbpel/2.0/.

[4] F. Baader, C. Lutz, M. Milicic, U. Sattler, and F. Wolter. Integrating description logics and action formalisms: First results. In *Association for the Advancement of Artificial Intelligence*, 2005.

[5] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):279–298, 1997.

[6] T. O. S. Coalition. OWL-S, 2003.

[7] I. Constantinescu, B. Faltings, and W. Binder. Large scale, type-compatible service composition. In *International Conference on Web Services*, 2004.

[8] D. Fensel et al. *Enabling Semantic Web Services: The Web Service Modeling Ontology*. Springer-Verlag, 2006.

[9] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning, II: The DLVK system. *Artificial Intelligence*, 144(1-2):157–211, 2003.

[10] T. Eiter and G. Gottlob. On the complexity of propositional knowledge base revision, updates, and counterfactuals. *Artificial Intelligence*, 57(2-3):227–270, 1992.

[11] G. D. Giacomo, M. Lenzerini, A. Poggi, and R. Rosati. On the update of DL ontologies at the instance level. In *Association for the Advancement of Artificial Intelligence*, 2006.

[12] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 2001.

[13] J. Hoffmann, I. Weber, J. Scicluna, T. Kaczmarek, and A. Ankolekar. Combining scalability and expressivity in the automatic composition of semantic web services, 2008. Available at http://members.deri.at/∼joergh/papers/tr-icwe08.pdf.

[14] Z. Liu, A. Ranganathan, and A. Riabov. Planning for message-oriented SWS composition. In *Association for the Advancement of Artificial Intelligence*, 2007.

[15] C. Lutz and U. Sattler. A proposal for describing services with DLs. In *Description Logics*, 2002.

[16] H. Meyer and M. Weske. Automated service composition using heuristic search. In *Business Process Management*, 2006.

[17] S. Narayanan and S. McIlraith. Simulation, verification, automated composition of web services. In *World Wide Web*, 2002.

[18] J. Pearl. *Heuristics*. Morgan Kaufmann, 1983.

[19] J. S. Penberthy and D. S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, pages 103–114, 1992.

[20] M. Pistore, P. Traverso, and P. Bertoli. Automated composition of web services by planning in asynchronous domains. In *International Conference on Automated Planning and Scheduling*, 2005.

[21] S. Ponnekanti and A. Fox. SWORD: A developer toolkit for web services composition. In *World Wide Web*, 2002.

[22] E. Sirin and B. Parsia. Planning for semantic web services. In *International Semantic Web Conference*, 2004.

[23] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan. Automated discovery, interaction and composition of semantic web services. *Journal of Web Semantics*, 1(1):27–46, 2003.

[24] I. Weber, I. Markovic, and C. Drumm. A Conceptual Framework for Composition in BPM. In *Business Information Systems*, 2007.

[25] M. Winslett. Reasoning about actions using a possible models approach. In *Association for the Advancement of Artificial Intelligence*, 1988.