A Syntactic Rule Based Approach to Web Service Composition

Ken Pu
University of Toronto
kenpu@cs.toronto.edu

Vagelis Hristidis
Florida International University
vagelis@cs.fiu.edu

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

Abstract

This paper studies a problem of web service composition from a syntactic approach. In contrast with other approaches on enriched semantic description such as state-transition description of web services, our focus is in the case when only the input-output type information from the WSDL specifications is available.

The web service composition problem is formally formulated as deriving a given desired type from a collection of available types and web services using a prescribed set of rules with costs. We show that solving the minimal cost composition is NP-complete in general, and present a practical solution based on dynamic programming. Experiements using a mixture of synthetic and real data sets show that our approach is viable and produces good results.

1 Introduction

Several efforts, including the Web Service Conversation Language (WSCL), the Business Process Execution Language for Web Services (BPELWS), and the integration of web service calls on XQuery [10] address the issue of web service composition. A natural problem is to automate the composition of web services. Towards this goal, a large corpus of work has focused on composing semantically rich (sematically annotated) web services. For instance, in [2, 5, 1] web services are modeled as finite-state machines that reflect their functionality, and their compositions are modeled as compositions of state machines. Such a semantically rich description of the web services allows discovery of high-quality compositions [2] as well as verification of existing ones [5]. However, WSDL, which is the currently widely adopted standard for description of web services, offers limited syntactic description of the services With only such limited information on the input-output types of the web services, is it possible to produce compositions to perform specific tasks?

In this paper, we propose a syntactic approach to web service composition, given only their WSDL descriptions. In particular, we view the web services as black boxes capable of transforming XML fragments of the input type to XML fragments of the output type, and hence the problem of web service composition becomes a type-derivation problem as follows: *Given a collection T_0 of base types, described in XML Schema, a collection S of web services, and a target type t, derive t from T_0 using the services in S.*

The target type is derived by applying a set of derivation-rules, or simply rules. These rules, which we will present in detail, describe the ways in which new types are derived from existing ones by application of web services. The target type is derived from the base types by a series of applications of these rules, which include sequential or parallel invocation of web services, and iteration over containers. Each step of a derivation (application of a rule) incurs a cost according to a cost model, which generally favors derivations with fewer rule applications (i.e., fewer structural transformations and web service applications).

Example: Consider the type `books` which is a collection of `book` elements. Each `book` element is identified either by its title or its ISBN number, along with the author names and the publisher.

$$\texttt{books}:\texttt{book}[1,\infty]:\left[\begin{array}{l} \textit{union}\left[\begin{array}{l}\texttt{title}:\texttt{string}\\ \texttt{ISBN}:\texttt{string}\end{array}\right]\ \text{or}\\ \texttt{authors}:\left[\begin{array}{l}\texttt{fname}:\texttt{string}\\ \texttt{lname}:\texttt{string}\end{array}\right]\\ \texttt{publisher}:\texttt{string}\end{array}\right.$$

Suppose that we have two web services described as follows:

$$\texttt{WS1}:\ \left[\begin{array}{l}\texttt{title}:\texttt{string}\\ \texttt{publisher}:\texttt{string}\end{array}\right]\longrightarrow\left[\begin{array}{l}\texttt{bestprice}:\texttt{numeric}\\ \texttt{bookstore}:\texttt{string}\end{array}\right]$$

$$\texttt{WS2}:\ \texttt{ISBN}:\texttt{string}\longrightarrow\left[\begin{array}{l}\texttt{bestprice}:\texttt{numeric}\\ \texttt{bookstore}:\texttt{string}\end{array}\right]$$

Suppose that one wishes to find the best prices of all the books in a collection (of type `books`). The target type is:

$$\texttt{bestprices}:\texttt{bestprice}[1,\infty]:\left[\begin{array}{l}\texttt{bestprice}:\texttt{numeric}\\ \texttt{bookstore}:\texttt{string}\end{array}\right]$$

One can manually verify that the target type `bestprices` can indeed be derived from the collection of base types $T_0 = \{\texttt{books}\}$ and web services $S = \{\texttt{WS1}, \texttt{WS2}\}$. The derivation is not unique, but a sensible one is the following.

Derivation Plan:
1. Iterate through the collection `books`,
2. if it is identified by its title, then
3. pass the title and publisher information to `WS1`,
4. else, if it is identified by its ISBN, then
5. pass the ISBN number to `WS2`.
6. Take the outputs and form the collection `bestprices`.

The rules proposed in this paper support this derivation as we provide rules for iteration (line 1), if-then-else on a type (line 2 and 4), type-construction and destruction (lines 3, 5 and 6), and application of web services (lines 3 and 5). We use a cost model to eliminate non-sensible derivations (such as using the author's last name as the publisher, and so on).

Naturally, the quality of the generated candidate compositions depends on the choice of the derivation rules. The set of rules used in this paper, which are shown to be sound experimentaly, are based on previous work on schema matching, type derivation and tree edit distance, which are areas related to this work as we explain in Section 2. However, our algorithmic framework is generic and a different set of rules can be specified according to the application needs [1]. In addition to the rules, the quality of the derivations is dependent on the naming quality of the WSDL files for the web services. Our case studies show that one can still find useful derivations using real-life WSDL files.

In this paper, we present our theoretical and algorithmic results on the *type-derivation problem*.

2 Related Work

Tree edit distance works [16, 21] (see [3] for a survey) define a minimal set of operations (typically add node, delete node, and relabel node) to tackle the problem of transforming a labeled tree T_1 to a labeled tree T_2 applying a minimum sequence of operations. Our problem is more complex since the labels of an XML Schema tree carry additional information (e.g., list, union, complex type, tag name, base-type, and so on) which makes plain relabeling inapplicable. For example, if two XML Schema trees differ only on the label of a single node, whose label is "union" for the first tree and "list" for the second, then they have unit distance in tree edit distance, but they are incompatible in our framework. Tree matching works [4] have the same limitation.

Dong *et al.* [6] propose a method to discover similar Web Services based on the textual descriptions (in WSDL) and their input/output parameters' names. It does not deal with composition however. Paolucci et al. [11] describe a framework to semantically match Web services described using DAML-S. Recent work has tackled the problem of composing web services by exploiting the flow diagrams that represent them. In particular, web services are viewed as state machines [20, 18, 1, 2].

Wombacher *et al.* [18] determines if a composition is valid by the intersection of the corresponding state machines. Furthermore, since the web services are modeled as transition structures, one can verify the web service composition by simulation [9] or by model-checking methods [5]. Such works, which are complementary to our approach, assume access to the behavioral characteristics of the service which may or may not be available; we only assume access to the service type signature which is typically available through WSDL (and XML Schema). In SWORD [12], web services are treated as input-output functions, and a set of rules of composition is specified. In our work, we also treat web services as functions, but work with a more complex set of rules that are capable constructing more sophisticated composition plans.

Schema matching tackles the problem of finding correspondences between the elements of two given schemas. Much work has been conducted on matching relational schemata [14], but recently there has also been work on matching of semi-structured schemata [13, 17, 15, 7, 8]. These algorithms match two schemata by matching their tree structures, exploiting possible semantic constraints [7, 17, 15]. In [8], the management of these mappings is considered. In contrast, our work tackles the problem of transforming a set of base types (schemata) to a target type using the available web services as transformation tools. The web services can be invoked in various ways during the transformation process according to a set of derivation rules.

3 The Optimal Type-Derivation Problem

In this section, we formally define the optimal type-derivation problem with respect to an arbitrary set of derivation rules. Then, we introduce a set of XML-centric rules which are used to capture the ways in which XML Schemas can be modified, combined, and web services invoked, in the derivation process. These rules naturally correspond to programming constructs found in BPEL, such as sequential and parallel invocations, iteration over lists, and branching based on node-types. We also included limited structural transformations, which can easily be implemented by XQuery, as part of the rule-set. Derivations using these rules correspond to compositions of web services coupled with possible structural transformations.

We represent an XML Schema as a tree of labeled nodes. The fragment of XML Schema we consider contains complex types of sequence, choice, and elements can have minOccurs and maxOccurs constraints. As a labeled tree, each node can be a tag name, a complex type constructor (`sequence`, `union`), or a primitive type such as `int` or `string`. A `union`-node represents a `choice`-complex type, that is, the instance of which can only be one of the children types in accordance with the XML Schema specification. Each node may optionally have a multiplicity modifier $[m,n]$ indicating that in the instance, its occurrence is between m and n inclusively – this corresponds to the minOccurs and maxOccurs constraints in XML Schema. We call a node *optional* if it is either a `union`-node, or a node with

[1] For instance, if we want to create more conservative compositions, we can remove the optional rule (described in Figure 3).

`minOccurs = 0` constraint. Types can, thus, be represented by terms such as `UNION`(title(string), `ISBN`(string)). A web service is simply a multi-input-multi-output function of the form $f : s_1, s_2, \cdots, s_n \rightarrow t_1, \cdots t_m$ where s_i and t_i are complex types.

PROBLEM FORMULATION: Let A_i and B be sets of types, called environments, and s_i and t types. We write $A_i \vdash s_i$ to mean that the type s_i can be derived from the environment A_i. Intuitively, an environment A_i is the collection of available types, and $A_i \vdash s_i$ is to say that, by some means, we can derive s_i from A_i. A *derivation-rule*, or simply a *rule*, has the form

$$\frac{A_1 \vdash s_1 \quad \cdots \quad A_n \vdash s_n}{B \vdash t}\ r.$$

It reads that if s_i can be derived from the environment A_i, then t can be derived from the environment B. The rule can be written as $r(B \vdash t)$ to indicate that r derives t from the environment B. A rule is grounded if it requires no $A_i \vdash s_i$, written $\frac{\emptyset}{B \vdash t}$. We further assume that each rule r carries a cost given by a generic cost function $\mathbf{cost}(r(B \vdash t))$. Derivation rules can be composed in the natural way to form derivations.

Formally, a *derivation* $D(B \vdash t)$ is defined as,

- any rule $r(B \vdash t)$ is a derivation, and
- if $r(B \vdash t)$ is a rule of the form $\frac{A_1 \vdash s_1 \quad \cdots \quad A_n \vdash s_n}{B \vdash t}\ r$, and $D_i(A_i \vdash s_i)$ are derivations, then

$$\frac{D_1(A_1 \vdash s_1) \quad \cdots \quad D_n(A_n \vdash s_n)}{B \vdash t}\ r \quad \text{is a derivation.}$$

- Nothing else is a derivation for $B \vdash t$.

A derivation $D(B \vdash t) = \frac{\{D_i(A_i \vdash t_i)\}_i}{B \vdash t} r$ can be viewed as a tree of rules, with the root being the rule $\frac{A_i \vdash t_i}{B \vdash t} r$, and sub-trees $D_i(A_i \vdash t_i)$. It says that type t is derived from the environment B by the derivation D. The derivation is grounded if its leaf-rules are all grounded. The cost for a derivation $D(B \vdash t)$, written $\|D(B \vdash t)\|$ is defined as,

- if $D(B \vdash t) = r(B \vdash t)$, then $\|D(B \vdash t)\| = \mathbf{cost}(r(B \vdash t))$,
- if $D(\vdash t) = \frac{D_1(A_1 \vdash s_1) \quad \cdots \quad D_n(A_n \vdash s_n)}{B \vdash t} r$, then

  $\|D(B \vdash t)\| = \mathbf{cost}(r(B \vdash t)) + g(\|D_1(A_1 \vdash s_1)\|, \ldots, \|D_n(A_n \vdash s_n)\|)$,

  where $g(\cdots)$ is either sum, or maximum [2] depending on the rule r.

Definition 1. *The* optimal type derivation problem *is: given a rule set R, a cost function* ***cost****, an environment A and a type t, find a grounded derivation* $D(A \vdash t)$ *with the minimal cost* $\|D(A \vdash t)\|$.

[2] In most cases, the cost of a derivation is the sum the costs of the sub-derivations and the applied rule. However, as defined in Figure 3, if the applied rule is *union*-rule, then we take the maximum of the costs of the sub-derivations.

The classical tree-edit distance problem [3] is a special instance of the optimal type derivation problem where the environment contains the source tree, and the derived type is the target tree. There are three rules: *insert*-, *delete*- and *replace*, each of unit cost. Tree edit-scripts correspond to derivations where the cost is always additive. The derivations, as defined, generalize tree-edit scripts in various ways. Derivations, in general, edit multiple trees, or types, in the environment to derive the new tree, and the rules allowed are a richer set that includes the basic tree-edit operations, but also ways of invoking web services.

From this point, we refer to types in the environment as the *base-types*, and the type to be derived as the *target type*.

3.1 Derivation rules for XML Schema

We use rules to express ways in which XML-documents (more accurately XML Schemas) can be transformed, combined and evaluated by web services. The set of rules include basic structural transformations (similar to the classical tree-edit operations), merging structures (substituting parts of a schema document with parts from other schema documents), and finally applying web services (evaluating a tuple of documents using available web services). In this framework, the web service composition problem is formally posed as a *type-derivation problem* in which the *base-types* are the available inputs, and the *target type* is the desired output. A derivation of the target type is then a plan of how the available web services can be composed, together with possibly some structural modifications to the intermediate results, to produce the desired output.

$$\frac{\emptyset}{A \vdash t}\ \text{MEM, if } t \in A. \qquad \frac{A_1 \vdash t_1 \quad A_2 \vdash t_2 \quad \cdots A_i \vdash t_n}{\bigcup_i A_i \vdash c(t_1, t_2, \ldots, t_n)}\ \text{CON}$$

$$\frac{A \vdash t}{A \vdash t|_p}\ \text{DES, where } p \text{ is not under an optional node.}$$

$$\frac{A \vdash t}{A \vdash t|_{\overline{p}}}\ \text{DEL} \qquad \frac{A \vdash t \quad A \vdash s}{A \vdash t \underset{p}{\leftarrow} s}\ \text{INS} \qquad \frac{A \vdash t \quad A \vdash s}{A \vdash t[s]_p}\ \text{REP}$$

Figure 1. Structural transformations

BASIC STRUCTURAL TRANSFORMATION RULES: The first set of rules deals with structural transformations. We refer to this set of rules as the *structural* rules, shown in Figure 1.

Membership: If type t is part of the environment, then it can be derived from the environment by the *membership*-rule (MEM).

Construct: One can derive new types from existing types by introducing new tags. Suppose one has derived `ISBN(string)` and `city(string)`, then certainly, we should be able to construct a new element `X(ISBN(string), city(string))` by introducing the new tag `X`. This is captured by the *construction*-rule (CON) shown in Figure 1.

Destruct: Contrary to the *construct*-rule, suppose that

$$\dfrac{\dfrac{\dfrac{\dfrac{\emptyset}{A\vdash \texttt{address}(\cdots)}\ \text{MEM}}{A\vdash \texttt{postal(string)}}\ \text{DES} \quad \dfrac{\dfrac{\emptyset}{A\vdash \texttt{person(age(int),name(string))}}\ \text{MEM} \quad \dfrac{\dfrac{\emptyset}{A\vdash \texttt{address}(\cdots)}\ \text{MEM}}{A\vdash \texttt{city(string)}}\ \text{DES}}{A\vdash \texttt{person(age(int),city(string))}}\ \text{REP}}{A\vdash \texttt{office(postal(string), person(age(int), city(string)))}}\ \text{CON}$$

Figure 2. A derivation plan.

we have the element `sequence(title(string), ISBN(string))`, then it is reasonable to derive `ISBN(string)` from it. Denote the sub-type (i.e., the sub-tree) of type t located from the root along a path p by $t|_p$. The *destruction*-rule (DES) derives the sub-type $t|_p$ of type t as long as $t|_p$ is not under some *optional* node. The reason that we do not allow applications of the *destruct*-rule at a sub-type under an optional node is simply that at the instance level, that sub-type may not be present. These sub-types are accessible by the complex rules presented in Figure 3.

Delete: The *delete*-rule (DEL) removes a sub-tree from a tree.

Insert: Similarly, one can insert a sub-type under a type as described by the *insert*-rule (INS). Denote the result of inserting type s into type t under the node located at path p as $t \underset{p}{\leftarrow} s$. If one can derive t and s, then application of the *insert*-rule derives $t \underset{p}{\leftarrow} s$.

Replace: The *replace*-rule (REP) allows one to replace a sub-type at path p of a type t with another derived type s, written $t[s]_p$.

This set of rules is not minimal in the sense that, for instance, the *replace*-rule is equivalent to composition of *insert* and *delete*-rules. However, depending on the cost function, it may be cheaper to replace rather than delete and then insert.

Example: Suppose the set of base-types is

$$A = \left\{ \texttt{address}: \left[\begin{array}{l}\texttt{city}:\texttt{string}\\ \texttt{postal}:\texttt{string}\end{array}\right], \texttt{person}: \left[\begin{array}{l}\texttt{age}:\texttt{integer}\\ \texttt{name}:\texttt{string}\end{array}\right] \right\}$$

And the target type t is $\texttt{office}: \left[\begin{array}{l}\texttt{postal}:\texttt{string}\\ \texttt{person}: \left[\begin{array}{l}\texttt{age}:\texttt{integer}\\ \texttt{city}:\texttt{string}\end{array}\right]\end{array}\right]$

One derivation $D(A\vdash t)$ is shown in Figure 2. In the plan, the target type is derived by replacing the sub-tree `name` under `person` with the sub-tree `city` which is derived by destructing the type `address`. Together, with the sub-tree `postal`, one uses construction to derive the final type. □

Note that the rules do not pay attention to the sequence of the children nodes. We actually ignore the order among the children nodes in the type, even though in general, XML Schema is order sensitive. The alternative is to introduce an additional structural rule, *permute*, which re-orders the children nodes under a given parent node. In this paper, we do not consider permutation, thus consider two nodes equal if their children are equal upto permutation.

$$\frac{A\vdash \texttt{union}(t_1,t_2,\cdots) \quad A\cup\{t_i\}\vdash t \text{ for all } t_i}{A\vdash t}\ \text{UNI}$$

$$\frac{A\vdash s_1 \quad \cdots A\vdash s_m \quad A\cup\{t_1,t_2\cdots t_n\}\vdash t}{A\vdash t}\ \text{APP}(f)$$

where there is a web service $f: s_1,s_2,\cdots s_m \to t_1,\cdots,t_n$.

$$\frac{A\vdash s[m,n] \quad A\cup\{s\}\vdash t}{A\vdash t[m',n']}\ \text{MAP where } m'\le m, n'\ge n.$$

$$\frac{\emptyset}{A\vdash s[0,n]}\ \text{OPTN}, \qquad \frac{s}{A\vdash s[m,n]}\ \text{LIST}, m\le 1$$

$$\frac{A\vdash c(t_1,t_2,\cdots,t_n) \quad A\vdash c(s_1,s_2,\cdots,s_m)}{A\vdash c(u_1,\ldots,u_{m+n})}\ \text{CAT}$$

where $\{u_i\}=\{s_i\}\cup\{t_i\}$ in bag semantics.

Figure 3. More complex XML-centric rules

DEALING WITH WEB-SERVICES AND XML-SCHEMA:
Clearly, structural rules are not suitable to lead to automation of web service compositions: they do not deal with important parts of XML Schema, in particular, `union` nodes in the type, and `minOccurs` and `maxOccurs` constraints of elements. Furthermore, they do not capture web services. Thus, we extend the rule set to include rules dealing with `union`'s, web services and finally dealing with `minOccurs`, `maxOccurs` constraints.

Union: The *union*-rule (UNI) reads as: if A can derive $\texttt{union}(t_1,t_2,\cdots)$ and for all $i\le n$, t_i together with A can derive t, then, A can derive t. Consider the following example: Let $A=\{\texttt{c}(\texttt{union}(\texttt{a}(\texttt{b}),\texttt{b}))\}$, and $t=\texttt{b}$. There is only one base-type. We cannot derive the target t from the base-type by structural-rules since destruction cannot be applied below a `union`-node. However, using the *union*-rule, one can derive t:

$$\frac{A\vdash \texttt{c}(\texttt{union}(\texttt{a}(\texttt{b}),\texttt{b})),\ A\cup\{\texttt{a}(\texttt{b})\}\vdash \texttt{b},\ A\cup\{\texttt{b}\}\vdash \texttt{b}}{A\vdash b}\ \text{UNI}$$

Apply: The *apply*-rule (APP) allows one to use an available web service to transform the tuple of input schemas to the output schema. It reads, if $f: s_1,s_2,\cdots\to t_1,t_2,\ldots$ is an available web service, and from the environment A, s_i are derivable, and A plus $\{t_j\}$ can derive t, then we can derive t from A using the web service.

For example, a web service $\texttt{getPrice}: \texttt{book},\texttt{ISBN}[0,1]\to\texttt{price}$ creates a rule

$$\frac{A\vdash \texttt{book} \quad A\vdash \texttt{ISBN}[0,1] \quad A\cup\{\texttt{price}\}\vdash t}{A\vdash t}\text{APP}.$$

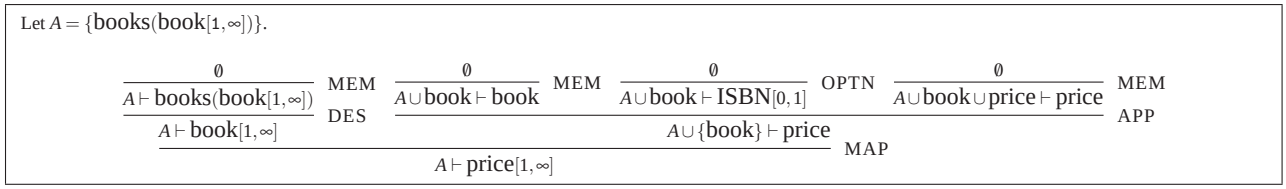


Figure 4. A derivation using *apply*, *optional* and *map*-rules.

Map, Optional and List: The rules *map* (MAP), *optional* (OPTN) and *list* (LIST) deal with elements with minOccurs and maxOccurs constraints. MAP allows one to map a list of elements of type s into a list of elements of type t given that t can be derived from s and the other base-types, and that the multiplicity constraint on t is more relaxed than the constraint for s. OPTN and LIST allow the creation of an element with multiplicity constraints from one without.

An example of using some of these rules along with the web service `getPrice` described above to derive the target type $t = \texttt{price}[1,\infty]$ from the base-type $\{\texttt{books}(\texttt{book}[1,\infty])\}$ is shown in Figure 4.

CAT is the *catenate*-rule which allows catenation of children of two nodes if they have the same label.

COSTS OF DERIVATIONS: The cost of each rule is given by a generic cost function **cost**. The cost function also determines the overall cost of a derivation. Given a derivation $D = \frac{D_1, D_2, \cdots D_n}{A \vdash t} r$, where $D_1, \ldots, D_n$ are also derivations, if the rule r is not the *union*-rule, then $\|D\| = \mathbf{cost}(r) + \sum_{i=1}^{n} \|D_i\|$. But in the case of the *union*-rule, we define $\|D\| = \mathbf{cost}(r) + \max\{\|D_i\| : 1 \le i \le n\}$.

THE COMPLEXITY OF TYPE-DERIVATION: The complexity of the type-derivation problem depends on the derivation rules we consider. In general, the problem is intractable. The source of intractability lies on the XML-centric rules in Figure 3.

Theorem 1. *If we consider only the union- and apply-rules, the decision problem of type-derivation is coNP-hard, and if we consider only the catenate-rule with constant cost, the optimal type derivation problem is NP-hard.*

In the following section, a dynamic programming solution is presented, and is shown to be optimal in some special cases of the type-derivation problem. The algorithm performs recursive back-tracking search from the target type back to the base-types. It is shown that the back-tracking algorithm for solving the optimal type-derivation problem with respect to the structural-rules is in polynomial time. For the intractable cases, we bound the depth of back-tracking along only the problematic rules.

4 Type-Derivation Algorithms

We present a dynamic programming algorithm, which we will refer to as the Back Tracking (**BT**)-algorithm, for solving the type derivation problem. It starts with the target type, and back-tracks by considering the possible rules that could have been used until a complete derivation plan is found. Given the base-types A, and a target type t, we construct a set of derivation plans to be considered, denoted by $\mathcal{C}(A \vdash t)$, and an optimal plan is simply given by

$$D_{\mathcal{C}}^{\text{opt}}(A \vdash t) = \text{argmin}\big\{\|D\| : D \in \mathcal{C}(A \vdash t)\big\}, \qquad (1)$$

where $\|D\|$ is the cost with respect to the cost model used. The definition of $\mathcal{C}(A \vdash t)$ will recursively make use of $D^{\text{opt}}(A' \vdash t')$ of some environment A' and type t'. The properties of the algorithm, such as its guaranteed termination, time complexity, and optimality, all depend on the choice of $\mathcal{C}(A \vdash t)$. Therefore, the computation for $D_{\mathcal{C}}^{\text{opt}}(A \vdash t)$ is completely characterized by the definition of the consideration set $\mathcal{C}(A \vdash t)$ and the choice of the cost model. The structure of the **BT**-algorithm is as follows.

```
opt_plan(A:base-types,t:target type)
  | C = consider(A,t);
  | if(C = ∅) return NON_DERIVABLE;
  | else return minimal plan in C;
consider(A:base-types, t:target type)
  | construct C with recursive calls to opt_plan();
  | return C;
```

In the subsequent sections, we describe the details of procedure `consider()`.

First we show how the case of structural-rules can be solved by this approach in polynomial time with a choice of $\mathcal{C}(A \vdash t)$ which will be described and analyzed in detail. Then it is extended to encompass the rest of the rules.

DEALING WITH STRUCTURE-TRANSFORMATION RULES: Let us restrict our attention only to the set of structural-transformation rules described in Figure 1. We first construct a set of maximal common-prefix embeddings which are defined below. Each embedding corresponds to a derivation, and the derivations of the maximal common-prefix embeddings form the consideration set needed in the **BT**-algorithm.

Recall that we view types as trees. The *tree domain* $\text{dom}(t)$ of a type t is simply all the paths of the nodes in t. Let s and t be two type trees. We define a *partial common-prefix embedding* between s and t to be a partial function $\theta : \text{dom}(s) \to \text{dom}(t)$ such that it satisfies the following condition:

- $\theta(0) = 0$, i.e., it maps the root of s to the root of t.
- $\forall p \in \text{dom}(s),\ \theta(p)$ is defined $\implies \text{label}(p) = \text{label}(\theta(p))$, i.e., θ maps only nodes of s to nodes of t with the same label.
- $\forall p \in \text{dom}(s),\ \theta(p)$ is defined $\implies \theta(\text{parent}(p)) = \text{parent}(\theta(p))$, i.e., θ preserves the parent-child relation.

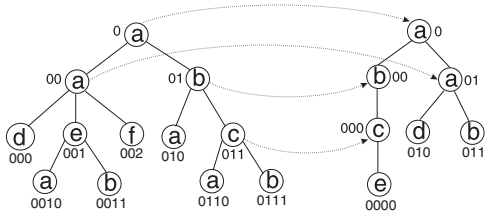


Figure 5. A partial common-prefix embedding from s (left) to t (right).

Partial common-prefix embeddings are henceforth referred to simply as embeddings. Computing embeddings between s and t can be done recursively in a straightforward way: start at the root nodes of s and t, stop if they differ in labeling or multiplicity, otherwise, build all possible matchings of the children nodes, and continue with the matched pairs.

The embedding θ is similar to the edit-script mapping found in tree-to-tree editing [3]. However, here, we do not allow relabeling tag names of XML elements in one operation. In order to perform this, one must first derive the children of the element (e.g. destruction), followed by a construction using the new tag name.

The domain of θ is $\mathrm{dom}(\theta) = \{p \in \mathrm{dom}(s) : \theta(p) \text{ is defined}\}$, and the range of θ is $\mathrm{rng}(\theta) = \{\theta(p) : p \in \mathrm{dom}(\theta)\}$. For the same types s,t, we say that one embedding θ_1 is greater than another θ_2 if $\mathrm{dom}(\theta_1) \supset \mathrm{dom}(\theta_2)$. The non-embedded nodes of s are $\mathrm{dom}(s) - \mathrm{dom}(\theta)$, and similar for t are $\mathrm{dom}(t) - \mathrm{rng}(\theta)$. The *frontier* of θ in s, $\mathrm{Fr}(\theta|s)$ is the most upper non-embedded nodes in s. The frontier $\mathrm{Fr}(\theta|t)$ in t is similarly defined.

Example: Consider the two types s and t shown in Figure 5. The embedding θ in Figure 5 is $\theta = \{0 \mapsto 0,\ 00 \mapsto 01,\ 01 \mapsto 00,\ 011 \mapsto 000\}$. Its domain and range are $\mathrm{dom}(\theta) = \{0, 00, 01, 011\}$, $\mathrm{rng}(\theta) = \{0, 00, 01, 000\}$. The frontiers are $\mathrm{Fr}(\theta|s) = \{000, 001, 002, 010, 0110, 0111\}$ and $\mathrm{Fr}(\theta|t) = \{0000, 010, 011\}$. □

Next we show how to recursively construct a derivation from an embedding. Assuming that we already have the optimal derivations $D^{\mathrm{opt}}(A \vdash s)$ and $\{D^{\mathrm{opt}}(A \vdash t|_p) : p \in \mathrm{Fr}(\theta|t)\}$ with some environment A, we form the plan $D_{s,\theta}(A \vdash t)$ that simulates the procedure in Figure 6. The derivation plan $D_{s,\theta}(A \vdash t)$ works as follows. First it generates s using $D^{\mathrm{opt}}(A \vdash s)$ (line 1). Next, $D_{s,\theta}$ replaces top unwanted nodes in s by nodes in t that share the same parent (lines 5-6). Then it adds the rest of the top nodes in t that are not already in s after the replacements (line 7). Finally, it removes all the unwanted nodes in s not already replaced (lines 11-13). It is easy to see that $D_{s,\theta}(A \vdash t)$ only needs *replace*, *delete* and *insert*-rules and the derivations $D^{\mathrm{opt}}(A \vdash s)$, $D^{\mathrm{opt}}(A \vdash t|_p)$ for $p \in \mathrm{Fr}(\theta|t)$.

Example: For the embedding of Figure 5, the frontiers are $\mathrm{Fr}(\theta|t) = \{t.0000, t.010, t.011\}$ and $\mathrm{Fr}(\theta|s) = \{s.000, s.001, s.002, s.010, s.0110, s.0111\}$.

```
1   derive s from A using D^opt(A ⊢ s)
2   for each p ∈ Fr(θ|t),
3     derive t|p from A using D^opt(A ⊢ t_p)
4     if childn(θ^-1(parent(p))) ∩ Fr(θ|s) ≠ ∅, then
5       pick one from childn(θ^-1(p)) ∩ Fr(θ|s), say q,
6       replace q with t|_p in s to derive s[t|_p]_q.
7     else
8       add t|_p under the node θ^-1(parent(p)) in s.
9     end if
10  end for
11  for each q ∈ Fr(θ|s) not deleted or replaced,
12    delete the sub-type q from s.
13  end for
```

Figure 6. The procedure equivalent to $D_{s,\theta}(A \vdash t)$.

For $p = t.0000$, $\mathrm{parent}(p) = t.000$. So $\theta^{-1}(\mathrm{parent}(p)) = s.011$, and $\mathrm{childn}(\theta^{-1}(\mathrm{parent}(p))) \cap \mathrm{Fr}(\theta|s) = \{s.0110, s.0111\}$ which is non-empty. Therefore, according to the procedure in Figure 6, $D_{s,\theta}(t)$ non-deterministically picks $s.0110$ to be replaced by $t.0000$. The sub-type $s.0111$ is deleted since it is not replaced by anything.

This is repeated for all the other nodes in $\mathrm{Fr}(\theta|t)$, and to summarize, we get a plan $D_{s,\theta}(A \vdash t)$ as follows.

1. Replace $s.0110$ with $t.0000$, $s.000$ with $t.010$, $s.001$ with $t.011$.
2. Remove $s.002$, $s.010$ and $s.0111$.
3. No inserts are needed in this case. □

At last, we construct the consideration set. The plans $D_{s,\theta}(t)$ provide the building blocks to the consideration set for the structural-rule set. Given the base-types in the environment A, and the target type $t = c(t_1, t_2, \cdots, t_n)$, the `consider()` procedure constructs the consideration set as follows:

$$\mathcal{C}^{\mathrm{STR}}(A \vdash t) = \left\{ \frac{D^{\mathrm{opt}}(A \vdash t_1)\ \cdots\ D^{\mathrm{opt}}(A \vdash t_n)}{A \vdash t} \mathrm{CON} \right\} \cup \left\{ D_{s,\theta}(t) : s \text{ is a subtype in } A \text{ not under optional-node and } \theta \in \mathrm{CP}(s,t) \right\}$$

where $\mathrm{CP}(s,t)$ is the set of all maximal partial common-prefix embeddings from s to t. In $\mathcal{C}^{\mathrm{STR}}(A \vdash t)$, we consider the case that the type t is derived by construction from its children t_i (each of which is optimally derived by $D^{\mathrm{opt}}(A \vdash t_i)$), and the cases in which t is derived from a subtype s in A by structural modification, which are the plans $\{D_{s,\theta}(A \vdash t)\}$.

The derivation of s in $D_{s,\theta}(t)$ simply consists of applications of the *membership*- and *destruct*-rules. Note that the construction of $D_{s,\theta}(t)$, and thus the definition of $\mathcal{C}^{\mathrm{STR}}(A \vdash t)$ make use of $D^{\mathrm{opt}}(A \vdash t)$. So computation for $D^{\mathrm{opt}}(A \vdash t)$ is recursive.

Proposition 1. *If the children of any node in A have distinct labels, then the recursive computation of $D^{\mathrm{opt}}(A \vdash t)$ using $\mathcal{C}^{\mathrm{STR}}(A \vdash t)$ terminates in polynomial time.*

Proposition 2. *The recursive computation for $D^{\mathrm{opt}}(A \vdash t)$ based on $\mathcal{C}^{\mathrm{STR}}(A \vdash t)$ terminates and is the optimal derivation if only structural-transformation rules with constant costs are allowed.*

An immediate corollary is that the optimal derivation problem with respect to the structural-rules with constant

costs that satisfies the distinct labels among siblings can be solved exactly in polynomial time.

DEALING WITH THE COMPLEX RULES: To incorporate the complex-rules, we augment the consideration set $\mathcal{C}(A \vdash t)$ to incorporate the *union*-, *apply*-, *optional*- *map*- and the *catenate*-rules.

$$\mathcal{C}^{\text{UNI}}(A \vdash t) = \left\{ \frac{D^{\text{DES}}(A \vdash \texttt{union}(t_i)) \quad D^{\text{opt}}(A \cup \{t_i\} \vdash t)}{A \vdash t}\text{UNI} \right\},$$

where $\texttt{union}(t_1, t_2, \cdots)$ are the union sub-types found in A that are not under some other union. The derivation $D^{\text{DES}}(A \vdash \texttt{union}(t_1, t_2, \ldots))$ is simply to obtain the sub-type $\texttt{union}(t_1, t_2, \ldots)$ from A using the *membership* and *de-struct*-rules.

$$\mathcal{C}^{\text{APP}}(A \vdash t) = \left\{ \frac{D^{\text{opt}}(A \vdash s_i) \quad D^{\text{opt}}(A \cup \{t_j\} \vdash t)}{A \vdash t}\text{APP} \right\}$$

for each available web service $f : s_1, s_2, \cdots \to t_1, t_2, \ldots$ where t is one of the output types.

If the target type is a type with multiplicity constraints, i.e., of the form $t[m,n]$, then consider also $\mathcal{C}^{\text{MAP}}$ and $\mathcal{C}^{\text{OPT}}$:

$$\mathcal{C}^{\text{MAP}}(A \vdash t[m,n]) = \left\{ \frac{D^{\text{DES}}(A \vdash s[m',n']) \quad D^{\text{opt}}(A \cup \{s\} \vdash t)}{A \vdash t[m,n]}\text{MAP} \right\}$$

where $s[m',n']$ are nodes with minOccurs and maxOccurs constraints in A that are not under an optional node and with $m \le m'$ and $n' \le n$.

We also consider all ways that $t[m,n]$ can be created by the *optional*- and *list*-rules.

$$\mathcal{C}^{\text{LIST}}(A \vdash t[m,n]) = \begin{cases} \left\{ \frac{\emptyset}{t[0,n]}\text{OPTN}, \frac{D^{\text{opt}}(A \vdash t)}{t[0,n]}\text{LIST} \right\} & \text{for } m = 0, \\ \left\{ \frac{D^{\text{opt}}(A \vdash t)}{t[0,n]}\text{LIST} \right\} & \text{for } m = 1, \\ \emptyset & \text{else.} \end{cases}$$

Thus, the augmented consideration set is:

$$\mathcal{C} = \mathcal{C}^{\text{STR}} \cup \mathcal{C}^{\text{UNI}} \cup \mathcal{C}^{\text{APP}} \cup \mathcal{C}^{\text{MAP}} \cup \mathcal{C}^{\text{LIST}}.$$

A straightforward recursive computation of $D^{\text{opt}}(A \vdash t)$ does not terminate using the new consideration set since the same rule can potentially be applied repeatedly. The solution is to maintain a set of avoidance rule set $\mathcal{X}$ during the back-tracking recursion – these are the rules not to be applied. So, during the back-tracking, we compute an optimal plan $D(A \vdash t | \neg \mathcal{X})$ that does not use any rules in $\mathcal{X}$. The recursive definition for the consideration sets become, for instance for $\mathcal{C}^{\text{APP}}$: Let $\mathcal{X}'_f = \mathcal{X} \cup \{\text{APP}(f)\}$ be the new avoidance set that avoids applying the web service f.

$$\mathcal{C}^{\text{APP}}(A \vdash t | \neg \mathcal{X}) = \left\{ \frac{D^{\text{opt}}(A \vdash s_1 | \neg \mathcal{X}'_f) \quad D^{\text{opt}}(A \vdash s_2 | \neg \mathcal{X}'_f) \cdots}{A \vdash t}\text{APP} \right\}$$

The avoidance set is augmented in a similar way when we back-track along the other rules. The avoidance set increases strictly during recursion, so the evaluation of $D^{\text{opt}}(A \vdash t | \neg \emptyset)$ is guaranteed to terminate, but in general in exponential time, which is clearly not acceptable in practice.

The exponential blow-up is due to the branching when back-tracking along the *union*- and *apply*-rules. So, we simply terminate the back-tracking if the number of avoided web services or unions reach their respective bounds. This

```
     N_ws^max : the bound on service composition,
     N_uni^max : the bound on number of union-rules.
 1   opt_plan(A, t, X)
 2   |  if num of APP in X > N_ws^max or
 3   |     num of UNI in X > N_uni^max then
 4   |    return NON-DERIVABLE;
 5   |  end if
 6   |  C = consider(A, t, X);
 7   |  return minimal cost plan in C;

 8   consider(A, t, X)
 9   |  C = C^STR(A ⊢ t) ∪ C^UNI(A ⊢ t|¬X)
     |                   ∪ C^APP(A ⊢ t|¬X);
10   |  if t = t'[m,n] then
11   |    C = C ∪ C^MAP(A ⊢ t) ∪ C^LIST(A ⊢ t)
12   |  end if
13   |  return C
```

Figure 7. The **BT-bnd**-algorithm

corresponds to finding a derivation plan that does not make use of consecutively composed web services of length greater than the bound. The variation of the **BT**-algorithm with bounded back-tracking will be referred to as the **BT-bnd**-algorithm. The overall **BT-bnd**-algorithm is shown in Figure 7. Note, if we remove lines 2–5 in Figure 7, we get back the **BT**-algorithm.

It is possible to extend the **BT**-algorithm to deal with more rules in case additional derivations rules become available. For instance, we have not included the *catenate*-rule as part of the search, but it can be easily added by, yet again, augmenting the consideration set $\mathcal{C}(A \vdash t)$. Given $t = c(t_1, t_2, \cdots, t_n)$, one can consider all plans of the form

$$\frac{D^{\text{opt}}(A \vdash c(t_{i_1}, t_{i_2}, \cdots)) \quad D^{\text{opt}}(A \vdash c(t_{j_1}, t_{j_2}, \cdots))}{A \vdash c(t_1, t_2, \cdots, t_n)}\text{CAT},$$

where $\{i_1, i_2, \cdots\} \cup \{j_1, j_2, \cdots\} = (1..n)$. Of course, there are exponentially many such plans to consider, so a similar bounded search heuristics is needed. We expect that this can be done for a great deal of other potentially useful rules.

AN APPLICATION OF THE ALGORITHM: We present sample results from the application of our techniques on real services. We have applied the **BT**-algorithm to a group of real internet web services described in `ws.strikeiron.com/USDAData?WSDL`. This group of web services provide access to a database of nutrient content of various food items. There are five services available offering searching and querying access to the database. Due to space limitation, only the relevant types and operations in the WSDL file are shown in condensed form in Figure 8.

A useful service is to retrieve nutrient information by keyword search. This service is not immediately offered. However, the operation `SearchFood` provides keyword search service but returns `NDBNumbers` as part of its output type `SearchFoodByDescriptionResult`, and operation `CalcNutrients` accepts NDBNumber as part of its input type and returns the nutrient content in its output type. We let the base-types be $A = \{\texttt{FoodKeywords}(\texttt{str}), \texttt{License}(\texttt{str})\}$, and the target $t = \texttt{MyNutrient}[0, \texttt{inf}](\texttt{NutrientValues}[0, \texttt{inf}])$. Each

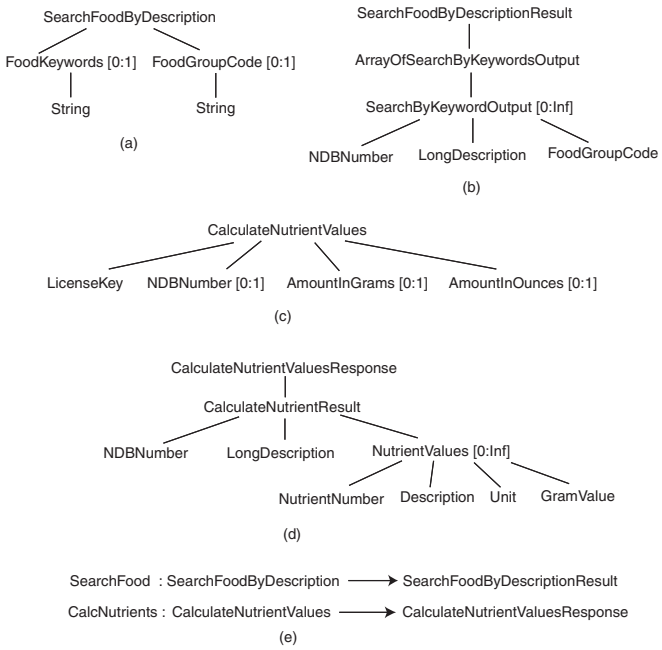


Figure 8. The types and operations in the WSDL file

`MyNutrient` element consists of a list of `NutrientValues` corresponding to the nutrients in some given food item. A derivation $D(A \vdash t)$ is a composition that allows one to obtain a collection of `MyNutrient` elements from a keyword represented by `FoodKeywords`.

The cost function $\mathbf{cost}(r(A \vdash t))$ that is used assigns a cost to each rule r and is insensitive to the environment A and the target type t. We assign a heavy cost to penalize the *construct*-rule and the *optional*-rule, a small cost to favor web services' *apply*-rule, and equal cost to all other rules. This cost-model reflects the preference of derivations for using composition of web services to produce the target type rather than using the structural-rules. One can explore more complicated cost-functions depending on the application, but in our experiments, this rather simple choice of the cost function allows the **BT**-algorithm to successfully report correct derivations; in this case $D(A \vdash t)$, makes use of the two web services and some structural construction and destruction. Similar results were obtained for other web service scenarios as well; we omit them, as well as detailed traces of the execution of the algorithm, due to space constraints.

5 Experimental evaluation

This section describes in detail the set of experiments performed to evaluate the performance of the algorithms. We have implemented the **BT**- and **BT-bnd**-algorithm based on the recursive definition of $D^{\text{opt}}(A \vdash t|\neg\mathcal{X})$, and $\mathcal{C}(A \vdash t|\neg\mathcal{X})$. Some optimizations in the search are possible. For instance, when backtracking along the *union*-rule in $\mathcal{C}^{\text{UNI}}$, we test to see if $D^{\text{opt}}(A \cup \{t_i\} \rightarrow t)$ ever explicitly uses t_i in deriving t. If not, then one concludes that the environment A is sufficient for deriving t optimally, thus, the algorithm can safely avoid considering deriving t from $\texttt{union}(t_i)$ using the *union*-rule.

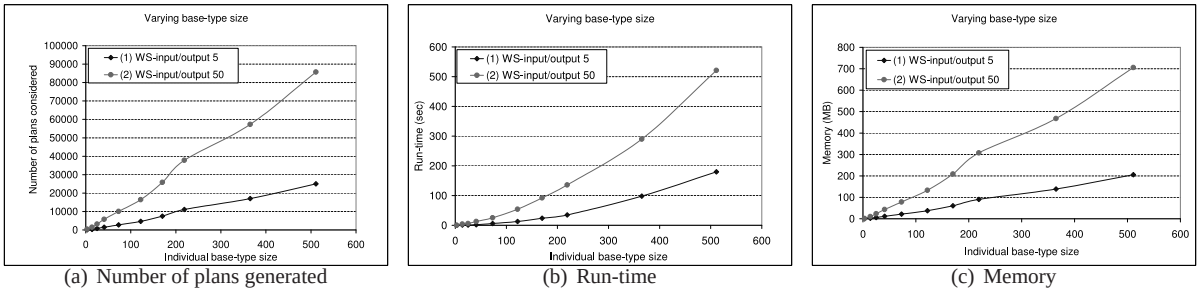
In order to improve efficiency, we amortize the recursion calls by maintaining an index of intermediate environments A, types t and the avoidance set $\mathcal{X}$ and the corresponding derivation $D^{\text{opt}}(A \vdash t|\neg\mathcal{X})$.

The parameters that are varied are the size of the base-types and the target type, the number of web-services, the number of union-nodes found in the base-type, and finally the number of minOccurs, maxOccurs constraints in the base-types. We examine the scalability of the algorithm in terms of the number of derivation plans considered during the search, the total run-time, and the memory consumption. The base-types are generated based on the schemas specified in the benchmark XBench [19]. However, in order to explore the scalability of the algorithm, for larger datasets and have ample flexibility in varying parameters of interest, we introduce additional nodes to these schemas. The additional tag names are randomly sampled from a fixed alphabet set, and the number of children of each node is randomly determined to be between 3-10. These experiments are carried out on a SunFire V440 server running Solaris 8 with SPECInt2k rated at 703 and SPECFP2k 1054.

SIZE OF INDIVIDUAL BASE-TYPES: We use 5 base-types based on the XBench schemas. The target type is a tree constructed based on the base-types by combining random sub-types from the base-types. The size of the target type is fixed at 50 nodes. There are no `union`-nodes, and there are 2 single-input web-services. The parameter we vary is the size of the individual base-types – from 10 nodes to 500 nodes. We also vary the size of the web-services' input-type and output-type from 5 to 50 nodes.

Figure 9(a) shows that the number of plans generated during the search has a polynomial growth with respect to the size of the base-types. Note that when the size of the individual base-types exceeds 200 nodes, the rate of growth changes from polynomial of higher power to nearly linear. The explanation is that since we have a fixed-size alphabet, repeated sub-types are found more frequently in the base-types, thus, more of the plans in the consideration set are found in the index, and do not need to be generated from scratch. Increasing the input/output type size for the web services uniformly increases the number of plans considered.

Figure 9(b) shows the run-time of the search with respect to the individual base-type size. The run-time increases polynomially with respect to the individual base-type size. Note that it does not display the near-linear behavior for larger base-types as was the case for the number of plans considered. This is due to the fact that the overall run-time includes time for both building the common-prefix embeddings *and* build the sub-plans. While the index helps in generating the sub-plans, building the common-prefix embedding still requires traversing through the base-types, and



(a) Number of plans generated (b) Run-time (c) Memory

Figure 9. Performance with increasing base-type size

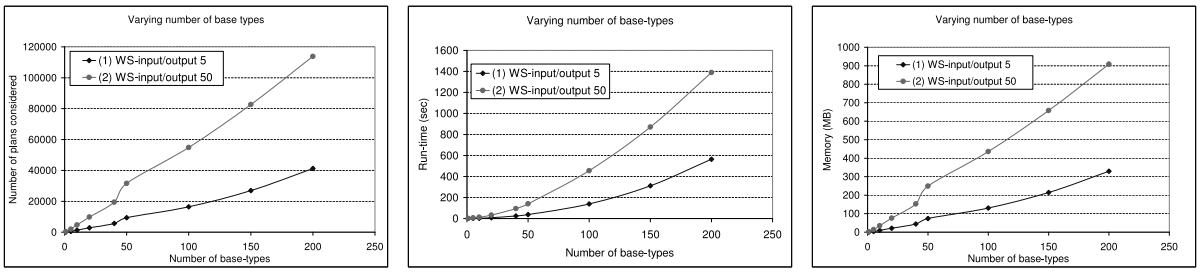


Figure 10. The performance with increasing number of base-types

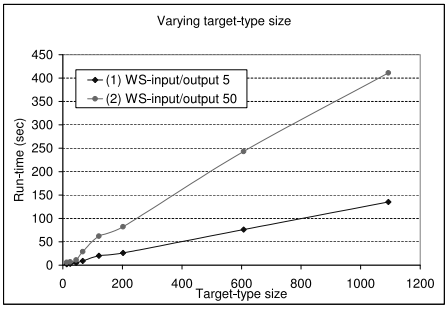


Figure 11. Performance with increasing target size

remains dominant.

Figure 9(c) shows the total memory used during the search. Note that it is proportional to the number of plans generated. When the web-services have larger input-output types, the **BT**-algorithm needs to consider more plans, thus taking time and memory. However, the trend remains consistent.

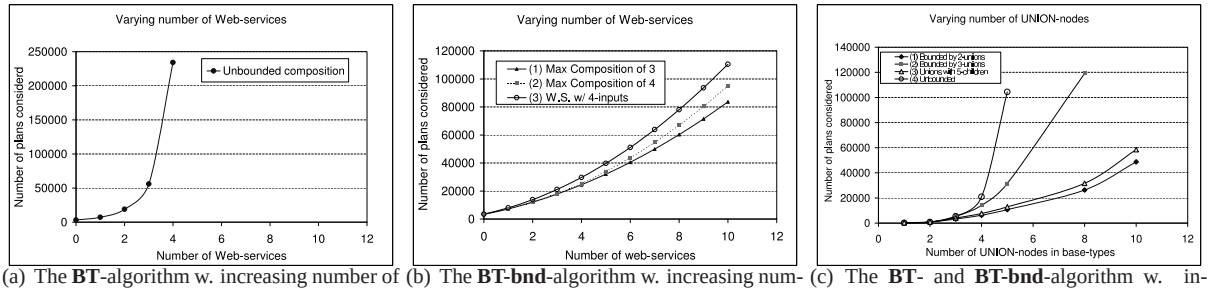
NUMBER OF BASE-TYPES: We also measure the performance with respect to the number of base-types. The number of base-types is increased from 1 to 200 while the size is fixed at 20 nodes. The other parameters remain unchanged. The number of plans generated, run-time and memory usage are shown in Figure 10. We see again the effect of the changing rate of growth for the number of plans generated. This is due to exactly the same reason as in Figure 9(a).

From Figure 9 and Figure 10, we verify that the performance is polynomial with respect to the size of the set of base-types, and is quite capable of handling large base-types, as well as a large number of them.

SIZE OF TARGET TYPE: For the next set of measurements, we fix the size of base-types to 20 nodes, and utilize 10 base-types for derivation. Again, the number of web-services is fixed at 2. We vary the target type size from 10 to 1000 to explore the performance when deriving a very large target type. The number of plans generated and the run-time are shown in Figure 11. Observe, that unlike the case of increasing base-types, the run-time becomes nearly linear when the target type size exceeds the threshold of 200. Since the target is so much larger than the base-types and the outputs of the web-services, construction time for the common-prefix tree is determined by the base-type size and no longer by the target type size. In conclusion, the **BT**-algorithm scales well with respect to the size of the target-type.

NUMBER OF WEB-SERVICES: We study the performance characteristics in terms of larger number of web-services. The base-types are fixed at 50 nodes, 5 base-types, and the target type is fixed at 50 nodes. The number of web-services is varied from 1 to 10. Each web-service has 2 input types, each with 5 nodes. We compare the performance of the **BT**-algorithm and **BT-bnd**-algorithm with different bounds on the composition length.

Figure 12(a) shows the number of plans considered by the **BT**-algorithm. As was shown in Section 4, an exponential number of plans are generated. By bounding the length of the composition to 3, only a polynomial number of plans are generated as it is shown in Figure 12(b) (1). Also in Figure 12(b) (2), we show the effect of increasing the length of the bound from 3 web-services to 4; it is evident that the performance implication is marginal and the overall trend is low polynomial. We also change the number of input types



(a) The **BT**-algorithm w. increasing number of WS (b) The **BT-bnd**-algorithm w. increasing number of WS (c) The **BT**- and **BT-bnd**-algorithm w. increasing unions

Figure 12. The performance of **BT**- and **BT-bnd**algorithms

to 4, and bound the composition length to 3 as shown in the same figure (3), again observing a smooth performance implication. The absolute run-time for the experiments in this figure are shown in seconds as follows:

Num of W.S.:	2	5	8	10
(2) Bnd-WS = 3, Num of Inp = 2	32	51	69	82
(3) Bnd-WS = 4, Num of Inp = 2	32	57	90	108
(4) Bnd-WS = 3, Num of Inp = 4	38	66	94	113

demonstrating very reasonable performance.

NUMBER OF `union`-NODES: Another source of exponential explosion for the search space is the number of union-nodes in the base-type. The next set of results show the performance with increasing number of union-nodes for the **BT**and **BT-bnd**-algorithms. We also vary the number of children of the union-nodes. The number of plans grows exponentially with respect to the number of unions in the unbounded case, as expected, and polynomially in the bounded case as shown in Figure 12(c). Corresponding absolute run time in seconds is as below.

Num of UNION-nodes	2	5	8	10
Bnd-UNI = 2, Num Children = 3	2	15	44	74
Bnd-UNI = 3, Num Children = 3	2	51	172	263
Bnd-UNI = 2, Num Children = 5	3	6	53	89

The experiments demonstrate that the algorithms are scalable with respect to the size of base-types and the target type, and sensitive to rules such as *apply* and *union* that require multiple backtracking matching our analytical expectation. These rules create an exponential explosion to the amount of search required. Placing bounds on the composition length and the number of union nodes to realistic values, make the problem tractable as demonstrated by our experiments.

6 Conclusions

We have presented a rule-based framework for web service composition. The problem of web service composition is seen as a optimal type derivation problem which is shown to be intractable in general. We have characterized a useful tractable class, and proposed suitable heuristics for the general case. The experimental results demonstrate the feasibility of our approach.

As part of on-going research, we are interested in experimenting with diverse cost models and rule sets, as well as incorporating semantics into our framework in order to improve the quality of the composition.

References

[1] B. Benatallah, Q.Z. Sheng, A.H.H. Ngu, and M. Dumas. Declarative composition and peer-to-peer provisioning of dynamic web services. In *ICDE*, 2002.

[2] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Massimo Mecella, and Rick Hull. Automatic composition of transition-based semantic web services with messaging. In *VLDB*, 2005.

[3] Philip Bille. Tree Edit Distance, Alignment Distance and Inclusion. *Technical report TR-2003-23 in IT University of Copenhagen*, 2003.

[4] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. In *SIGMOD*, 1997.

[5] A. Deutsch, L. Sui, and V. Vianu. Specification and Verification of Data Driven Web Services. *PODS*, 2004.

[6] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity Search for Web Services. *VLDB*, 2004.

[7] J. Madhavan, P. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. *Proceedings of VLDB*, 2001.

[8] S. Melnik, P. A. Bernstein, A. Halevy, and E. Rahm. Supporting executable mappings in model management. In *SIGMOD*, 2005.

[9] S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. *WWW*, 2002.

[10] N. Onose and J. Simeon. XQuery at Your Web Service. *WWW*, 2004.

[11] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia P. Sycara. Semantic matching of web services capabilities. In *ISWC*, 2002.

[12] Shankar R. Ponnekati and Armando Fox. SWORD: A developer toolkit for web service composition. In *WWW*, 2002.

[13] Lucian Popa, Yannis Velegrakis, Renee J. Miller, Mauricio A. Hernandez, and Ronald Fagin. Translating web data. In *VLDB*, 2002.

[14] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[15] Erhard Rahm, Hong-Hai Do, and Sabine Mabmann. Matching large XML schemas. *SIGMOD Record*, 33(4):26–31, 2004.

[16] Dennis Shasha and Kaizhong Zhang. Fast algorithms for the unit cost editing distance between trees. *J. Algorithms*, 11(4):581–621, 1990.

[17] Hong Su, Harumi Kuno, and Elke A. Rundensteiner. Automating the transformation of XML documents. In *WIDM*, 2001.

[18] A. Wombacher, Peter Fankhauser, B. Mahleko, and E. Neuhold. Matchmaking for business processes based on choreographies. In *IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE)*, 2004.

[19] B.B. Yao, M.T. Ozsu, and N. Khandelwal. XBench benchmark and performance testing of xml dbmss. In *ICDE*, pages 621–632, 2004.

[20] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality driven web services composition. In *WWW Conference*, 2003.

[21] Kaizhong Zhang, Jason T. L. Wang, and Dennis Shasha. On the Editing Distance between Undirected Acyclic Graphs and Related Problems. *International Journal of Foundations of Computer Science*, 1995.