Modeling and Synthesis of Service Composition using Tree Automata

Ying Zhu[1], Ken Q. Pu[2]
[1]Faculty of Business and Information Technology
[2]Faculty of Science
University of Ontario Inst. of Technology
2000 Simcoe Street North, Oshawa, ON, Canada, L1H 7K4
1-905-721-8668
Email: [1]ying.zhu@uoit.ca, [2]ken.pu@uoit.ca

Abstract— **We revisit the problem of synthesis of service composition in the context of service oriented architecture from a tree automata perspective. Comparing to existing finite state machine and graph-based approaches to the problem of service composition, tree automata offers a more flexible and faithful modeling of multi-input services and their admissible compositions. In our framework, tree automata is used to express both type signature constraints of individual services as well as constraints on the order in which services must be invoked. To synthesize service compositions, users may provide optional specifications on the desired composit service. The user specifications are also expressed as tree automata. We employee a combination of tree automata algorithms to compute the set of all possible valid service compositions which satisfy the user specifications.**

I. Introduction

In service oriented architecture (SOA), the emphasis is to expose functionality and business processes as services. It has been shown [1] that SOA offers the advantage of modularity, extensibility and flexibility to the development of large enterprise business process and information management systems. An important research topic in SOA that has attracted much attention is the problem of automatic service composition. The problem automatic service composition deals with fully automated or semi-automated algorithms to generate different ways of composing existing services into composit services which satisfie a set of given specifications. Service composition is now an increasingly important problem because of the popular adaptation of the SOA by major service providers. For instance, Google offers access to their web-based services such as calendar, documents, photo albums and blogs etc. as online web services[1], and major enterprises, such as Staples, have been building their intranet infrastructure based on the SOA[2]. Naturally, it is highly desirable if, algorithmically, one can discover compositions of existing services to form new composit services which offer new functionalities.

Services cannot be free composed due to many constraints. One type of constraint is the type signatures of the inputs and outputs of services. In this article, we treat services mappings which map multiple input values to an output value. The inputs and output of a service are typed by some type system, for example, the web service description language[3] (WSDL). In service compositions, the output of one service is passed to the one of the inputs of another service, and this is valid if and only if the types fo the output value and the input value of the two respective services agree. Disagreements of the input-output types are known as *type errors* in programming languages. All valid service compositions should be type error free. We refer to these constraints as *type signature constraints*.

Another constraint in how services can be composed is the order in which services are allowed to be invoked. Service providers typically restricts the sequence of invocation of services. For instance, all Google web services require the user to be authenticated before other web services can be invoked. Thus, any service composition that uses Google web services must invoke the Google authentication service before invoking other Google data services. We refer to these type of constraints as *invocation constraints*.

Finally, when computing service compositions, we also need to consider user specifications. The user speci-

[1]http://code.google.com/apis/gdata/

[2]http://www-306.ibm.com/software/solutions/soa/

[3]http://www.w3.org/TR/wsdl

fications further constraint the service compositions that we wish to synthesize.

In this article, we present a tree automata based framework for describing services that is capable of capturing all different type of constraints: type signature constraints, invocation constraints and the user specified constraints.

By applying tree automata algorithms, we are able to automatically generate all possible service compositions which satisfy the type signature contraints, the invocation constraints, and of course, the user specified constraints.

II. Related Work

Many different Web service composition methods have been proposed. A strong body of work has emerged that takes the approach of AI planning. We highlight here the most relevant work.

In [2], [3], the authors formulate user requests and Web services in the first-order language of the situation calculus (a logical language for reasoning about action and change); they adapt and extend the Golog language – a logic programming language built upon the situation calculus – to automatically construct composed Web services.

Medjahed *et al.* [4] proposes a method that begins with the specification of desired compositions by high-level declarative description (using a language called Composite Service Specification Language (CSSL)), and then uses composability rules to perform matchmaking between services which generates composition plans that satisfies the requester's specifications. The composability rules include both syntactic rules – compatibility of output and input messages of two services – and semantic rules – compatibility of domains, categories and purposes of two services.

Another proposed method that uses rule-based composition plan generation is SWORD [5]. Web services are specified by the Entity-Relationship (ER) model. A Web service is modeled as a Horn rule specifying that the postconditions of the service are achieved if its preconditions are true. The requester specifies the initial and gaol states of the composite service and the plan generation is done by a rule-based expert system.

Many of the previous work [2], [6], [7] present composition methods that work on semantic web service descriptions such as OWL-S or DAML-S. In these approaches, the services are modeled in terms of their inputs, outputs, preconditions and effects. Some of the others [8], [9] work on a process model defined in BPEL or as transition systems.

Recently, Liu *et al.* [10] presented a semantic model of web services using RDF graph patterns, giving rich semantic descriptions of input and output messages of the services. A key distinguishing feature of this model is semantic propagation – the semantic description of the output message depends on that of the input – and this is done through graph transformations. The model can be used to check if some services' output messages can be constructed into input messages to other services. Based on this, composition is done by representing the services as a workflow graph.

Some work focus on the matchmaking of web services for building compatible connections between them. In [11], composition is a directed graph whose nodes are linked by a matching compatibility – Exact, Subsume, PlugIn, Disjoint – between input and output parameters. A method is proposed by [12] that augment standard semantic matching functions with a computation of syntactic similarity. Both [13] and [14] exploited Concept Abduction so as to return non-exact composition results to the user request. An approximate orchestration of services is computed in [13], while the constraints from user request are relaxed in [14].

III. Modeling Services and Constraints Using Tree Automata

In this section we review the necessary background on tree automata and how tree automata can be used to model services and the various types of constraints.

A. Tree automata

Definition 1 (Tree Automaton): A tree automaton $\mathcal{A}$ consists of an alphabet Σ of symbols, a set of states Q, a set of final states $Q_F \subseteq Q$, and a transition relation Δ. Every symbol f in the alphabet Σ is a function name, and has an arity $\text{arity}(f) \geq 0$. The arity of f is simply the number of inputs it expects. Functions with arity equal to zero are called *constants*, written $a, b, c, \ldots$. A transition rule δ is of the form:

$$q_1, q_2, \ldots, q_n \xrightarrow{f} q$$

where $f \in \Sigma$, q and q_i are states in Q, and $n = \text{arity}(f)$. ■

For a constant $a \in \Sigma$, the transition rule δ becomes $\xrightarrow{a} q$. The transition relation Δ is simply a set of transition rules. We will refer to Σ as a functional alphabet to distinguish it from the alphabets in the theory of finite-state automata.

Definition 2 (Terms and Subterms): Given a functional alphabet Σ, the set of terms is defined as the smallest set $T(\Sigma)$ such that all constants in Σ belong to $T(\Sigma)$ and if $t_1, t_2, \ldots t_n \in T(\Sigma)$ and $f \in \Sigma$ with arity$(f) = n$, then the expression $f(t_1, t_2, \ldots, t_n) \in T(\Sigma)$.

Given a term $t \in T(\Sigma)$, its sub-terms $S(t)$ is defined recursively as:

- If $t = a$ where a is a constant in Σ, then $S(t) = \{a\}$.
- If $t = f(t_1, t_2, \ldots, t_n)$, then
$$S(t) = \{t, t_1, t_2, \ldots, t_n\} \cup \bigcup_i S(t_i)$$
- Nothing else belongs to $S(t)$.

■

Similar to classical finite state machines, a tree automaton $\mathcal{A}$ defines a subset of $T(\Sigma)$, known as the language of $\mathcal{A}$, written $\mathcal{L}(\mathcal{A})$. In order to define the $\mathcal{L}(\mathcal{A})$, we need to define the *acceptance relation*.

Definition 3 (Term Acceptance and Tree Language): Given a tree automaton $\mathcal{A}$, we define a function $\alpha : T(\Sigma) \to 2^Q$ which maps a term to a set of *reachable* states of $\mathcal{A}$. The function α is defined recursively:

- For constants $a \in \Sigma$,
$$\alpha(a) = \{q \in Q : \exists \delta \in \Delta \text{ such that } \delta = (\xrightarrow{a} q)\}$$
- For a term $t = f(t_1, t_2, \ldots, t_n)$,
$$\alpha(t) = \{q \in Q : \exists \delta \in \Delta, \exists q_1 \in \alpha(t_1), \ldots, \exists q_n \in \alpha(t_i) \text{ such that } \delta = (q_1, q_2, \ldots q_n \xrightarrow{f} q)\}$$

A term t is *accepted* by the automaton $\mathcal{A}$ if $\alpha(t) \cap Q_F \neq \emptyset$. The language $\mathcal{L}(\mathcal{A})$ of a tree automaton $\mathcal{A}$ is all the terms that are accepted by $\mathcal{A}$. ■

Example 1: Let $\mathcal{A}$ be a tree automaton with three state $Q = \{q_1, q_2, q_3\}$ and one final state $Q_F = \{q_3\}$. The alphabet has three functional symbols: $\{f(\_,\_), g(\_), a\}$. The transition rules are:

$$\xrightarrow{a} q_1$$
$$(q_1, q_2) \xrightarrow{f} q_3$$
$$q_1 \xrightarrow{g} q_2$$
$$q_2 \xrightarrow{g} q_2$$

The language $\mathcal{L}(\mathcal{A})$ is infinite. It consists of the following terms:

$$t_1 = f(a, g(a))$$
$$t_2 = f(a, g(g(a)))$$
$$t_3 = f(a, g(g(g(a))))$$
$$\cdots$$

Consider $t_1 = f(a, g(a))$. Its subterms are $S(t) = \{f(a, g(a)), g(a), a\}$. The reachable states $\alpha(g(a)) = \{q_2\}$. Since $Q_F \cap \alpha(g(a)) = \emptyset$, the term $g(a)$ is not part of the language $\mathcal{L}(\mathcal{A})$, whereas, $\alpha(f(a, g(a)) = \{q_3\}$ contains a final state, hence $f(a, g(a)) \in \mathcal{L}(\mathcal{A})$. ■

B. Modeling Services

In this section, we describe by a running case study how tree automata are used to model services and their various constraints.

Example 2 (A case study): Consider a college web site offering number of services to registered students. One service is lookup, lookup, the required text book based on the course code and a semester. Another service, bestbuy, locates a book store for a given text book. One may optionally specify a city to search within. The service, print, displays the description of either a text book or a book store. Finally the service, login, performs authentication using the specified user and password. The type signatures of the various services are shown in Figure 1. Note that the service print is polymorphic in the sense that it can accept two types of arguments as input, while the service bestbuy has an *optional* second argument. Also note that the output types of the services print and login are `void` which indicates that they are do not have a return value, but are *procedures* which have side-effects. The policy is that one must login first before invoking any other services.

We further introduce a number of constants. Physics is a course, Current is the current semester, LoginProfile stores the value for user and password. ■

In Example 2, there can be a number of possible ways of composing these services, some are valid, and others are not. For example, the following is a composition which displays the book store currently used by the Physics course:

```
login(LoginProfile.User, LoginProfile.Password);
print(bestbuy(lookup(Physics,Current)));
```

Consider the next composition which displays the information of the text book used by Physics:

```
print(bestbuy(lookup(Physics,Current)));
```

It is invalid because it violates the invocation constraint that login must be invoked before other services.

The next composition attempts to display the text book information used by the current Physics course.

```
login(LoginProfile.User, LoginProfile.Password);
print(lookup(Physics));
```

$$\begin{aligned}\textsf{lookup} &: \texttt{Course}, \texttt{Semester} \rightarrow \texttt{TextBook}\\ \textsf{bestbuy} &: \texttt{TextBook}, \texttt{City}? \rightarrow \texttt{BookStore}\\ \textsf{print} &: \texttt{TextBook}|\texttt{BookStore} \rightarrow \texttt{void}\\ \textsf{login} &: \texttt{User}, \texttt{Password} \rightarrow \texttt{void}\\ \textsf{Physics} &: \texttt{Course}\\ \textsf{Current} &: \texttt{Semister}\\ \textsf{LoginProfile} &: \texttt{User}, \texttt{Password}\end{aligned}$$

Fig. 1. Services and their signatures

Unfortunately, it violates the type constraint as the lookup service requires two arguments: a course and a semester.

In order to model the services shown in Figure 1, their type constraints and the invocation constraints, we propose to construct multiple tree automata – each tree automaton describes a separate type of constraints.

The Functional Alphabet

The functional alphabet consists of the service names: In addition, in order to model the invocation of multiple procedures, we introduce one more functional symbol

$$\textsf{do} : \texttt{void}, \texttt{void} \rightarrow \texttt{void}$$

Thus the alphabet is given by:

$$\Sigma = \{\textsf{lookup, bestbuy, print, login, Physics, Current, LoginProfile}\}$$

Modeling Type Constraints

We begin by constructing a tree automaton $\mathcal{A}_{\text{type}}$ which describes *all* type valid compositions. The states are all possible data types:

$$Q_{\text{type}} = \{\texttt{Course, Semester, TextBook, City, BookStore, User, Password}\}$$

The transition rules are quite similar to the functional signatures. The transition rules are shown in Figure 2

All states are final states for $\mathcal{A}_{\text{type}}$. It is straight forward to verify that terms in $\mathcal{L}(\mathcal{A})$ are all the compositions which are free of type errors. However, it does not eliminate the service compositions which fail to login first. For that, we need to construct another tree automaton to model the invocation constraints.

Modeling Invocation Constraints

We construct another tree automaton $\mathcal{A}_{\text{invoke}}$. The language $\mathcal{L}(\mathcal{A}_{\text{invoke}})$ must include all invocations sequences of procedures which starts with login. The states are $Q_{\text{invoke}} = \{q_1, q_2, q_3, q_4\}$. The transition rules for $\mathcal{A}_{\text{invoke}}$ is shown in Figure 3.

$$\begin{aligned}(\texttt{Course}, \texttt{Semester}) &\xrightarrow{\textsf{lookup}} \texttt{TextBook}\\ \texttt{TextBook} &\xrightarrow{\textsf{bestbuy}} \texttt{BookStore}\\ (\texttt{TextBook}, \texttt{City}) &\xrightarrow{\textsf{bestbuy}} \texttt{BookStore}\\ \texttt{TextBook} &\xrightarrow{\textsf{Print}} \texttt{void}\\ \texttt{BookStore} &\xrightarrow{\textsf{Print}} \texttt{void}\\ (\texttt{User}, \texttt{Password}) &\xrightarrow{\textsf{login}} \texttt{void}\\ (\texttt{void}, \texttt{void}) &\xrightarrow{\textsf{do}} \texttt{void}\\ &\xrightarrow{\textsf{Physics}} \texttt{Course}\\ &\xrightarrow{\textsf{Current}} \texttt{Semester}\\ &\xrightarrow{\textsf{LoginProfile}} \texttt{User}\\ &\xrightarrow{\textsf{LoginProfile}} \texttt{Password}\end{aligned}$$

Fig. 2. Transition rules for $\mathcal{A}_{\text{type}}$.

The final states are $Q_{F\text{invoke}} = \{q_4\}$.

It's easy to see that term in $\mathcal{L}(\mathcal{A}_{\text{invoke}})$ are all sequences of procedures that start with login: the only way to reach the final state of q_4 is the invoke login first to reach the state q_2, and then invoke do arbitrary number of times (to concatenate with other procedures) and reach q_4.

Terms in $\mathcal{L}(\mathcal{A}_{\text{invoke}})$ obeys invocation constraints, but may violate the type constraints. In Section III-C, we will discuss present the well-known facts regarding tree automata – the closure of regular tree languages. It is possible to construct an intersection tree automaton from $\mathcal{A}_{\text{type}}$ and $\mathcal{A}_{\text{invoke}}$ and with a language that respects both the type constraints and invocation constraints.

$$
\begin{aligned}
(q_1, q_1) &\xrightarrow{\textsf{lookup}} q_1\\
(q_1, q_1) &\xrightarrow{\textsf{bestbuy}} q_1\\
q_1 &\xrightarrow{\textsf{bestbuy}} q_1\\
&\xrightarrow{\textsf{Physics}} q_1\\
&\xrightarrow{\textsf{Current}} q_1\\
&\xrightarrow{\textsf{LoginProfile}} q_1\\
(q_1, q_1) &\xrightarrow{\textsf{login}} q_2\\
q_1 &\xrightarrow{\textsf{Print}} q_3\\
(q_2, q_3) &\xrightarrow{\textsf{do}} q_4\\
(q_3, q_3) &\xrightarrow{\textsf{do}} q_3
\end{aligned}
$$

Fig. 3. Transition rules for $\mathcal{A}_{\text{invoke}}$

MODELING USER SPECIFICATIONS

Users can impose specifications on the properties of the desired compositions. As we have already demonstrated in previous examples, tree automata are very flexible in expressing properties of terms. Users can express their specifications by construction tree automata.

Example 3: Suppose that the user wish to synthesize a composite service which prints out all possible information which is relevant to Physics course and Current semester. We simply construct a tree automaton $\mathcal{A}_{\text{user}}$ which describes *all* terms which are built using only the constants Physics, Current and LoginProfile. The following automton transition rules describe such terms:

$$
\begin{aligned}
&\xrightarrow{\textsf{Physics}} q_1\\
&\xrightarrow{\textsf{Current}} q_1\\
&\xrightarrow{\textsf{LoginProfile}} q_1\\
q_1 &\xrightarrow{x} q_1\\
(q_1, q_1) &\xrightarrow{x} q_1\\
q_1 &\xrightarrow{\textsf{Print}} q_2
\end{aligned}
$$

where x is an available service name: $\{\textsf{lookup}, \textsf{bestbuy}, \dots\}$. The final state is q_2. The language $\mathcal{L}(\mathcal{A}_{\text{user}})$ describe all compositions which are built using Physics, Currentand LoginProfile, and ends with Print.

In the next section we describe how well-known operations on tree automata can be used to automatically synthesize service compositions that satisfy the user specification $\mathcal{A}_{\text{user}}$, the type constraints $\mathcal{A}_{\text{type}}$ and the invocation constraints $\mathcal{A}_{\text{invoke}}$.

C. Computing the compositions

We demonstrated how tree automata offer flexible expressive power to describe various constraints on how services can be composed together, as well as user specifications. In this section, we present a number of known facts and algorithms [15].

Fact 1 (Intersection of Tree Automata): Given two tree automata $\mathcal{A}_1$ and $\mathcal{A}_2$. There always exists an tree automaton $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$. ■

Fact 2 (Minimization of Tree Automaton): Given an tree automaton $\mathcal{A}_1$, there exists a unique tree automaton $\mathcal{A}$ with minimal number of states such that $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A})$. ■

It is straight forward to apply the intersection algorithm and the minimization algorithm to generate service compositions.

Algorithm 1 ServiceComposition

```
$\mathcal{A}_{\text{type}}$ = type constraints
$\mathcal{A}_{\text{invoke}}$ = invocation constraints
$\mathcal{A}_{\text{user}}$ = user specifications
$\mathcal{A} = \mathcal{A}_{\text{type}} \cap \mathcal{A}_{\text{invoke}} \cap \mathcal{A}_{\text{user}}$
$\mathcal{A}^* = \text{minimize}(\mathcal{A})$
if States for $\mathcal{A}^*$ is empty then
  return error {no composition found}
else
  return $\mathcal{A}^*$ {all terms in $\mathcal{L}(\mathcal{A}^*)$ are potential compositions}
end if
```

IV. CONCLUSION AND FUTURE WORK

In this article, we described a tree automata framework to model services and their compositional constraints. As an extention to previous work on service composition, tree automata offer flexible expressiveness to describe constraints based on type signatures of services, as well as constraints on the order of invocation of services. By example, we also described how tree automata can be used to express user specifications on the desired service compositions. By utilitizing well-known operations on tree automata, we can easily enumerate all possible service composition plans which free of type errors and satisfy the user specifications and invocation constraints.

As future work, we wish to integrate the proposed tree automata based composition synthesis with complex type systems such as XML schema to reason about composition of web services. We also wish to extend the composition plans to include programming constructs such as branching and loop controls.

REFERENCES

[1] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

[2] S. Narayanan and S. McIlraith, "Simulation, verification and automated composition of web services," in *Proceedings of the 11th International World Wide Web Conference*, May 2002.

[3] S. McIlraith and T. C. Son, "Adapting golog for composition of semantic web services," in *Proceedings of the 8th International Conference on Knowledge Representation and Reasoning (KR2002)*, April 2002.

[4] B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid, "Composing web services on the semantic web," *The VLDB Journal*, November 2003.

[5] S. R. Ponnekanti and A. Fox, "Sword: A developer toolkit for web service composition," in *Proceedings of the 11th World Wide Web Conference*, 2002.

[6] D. Berardi, D. Calvanese, G. D. Giacomo, R. Hull, and M. Mecella, "Automatic composition of transition-based semantic web services with messaging," in *VLDB*, 2005.

[7] P. Traverso and M. Pistore, "Automated composition of semantic web services into executable processes," in *ISWC'04*, 2004.

[8] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi, "Automatic sythesis of composite bpel4ws web service," in *ICSOC*, 2006.

[9] J. Pathak, S. Basu, and V. Honavar, "Modeling web services by iterative reformulation of functional and non-functional requirements," in *ICSOC*, 2006.

[10] Z. Liu, A. Ranganathan, and A. Riabov, "Modeling web services using semantic graph transformations to aid automatic composition," in *IEEE International Conference on Web Services (ICWS 2007)*, 2007.

[11] R. Zhang and I. B. A. et al., "Automatic composition of semantic web services," in *ICWS*, 2003, pp. 38–41.

[12] J. Cardoso and A. Sheth, "Semantic e-workflow composition," *Journal of Intelligent Information Systems*, vol. 21, pp. 191–225, 2003.

[13] T. D. Noia and E. D. S. et al., "Automated semantic web services orchestration via concept covering," in *Proc. of the 14th International World Wide Web Conference (WWW05)*, 2005, pp. 1160–1161.

[14] F. Lecue, A. Delteil, and A. Leger, "Applying abduction in semantic web service composition," in *IEEE International Conference on Web Services (ICWS 2007)*, 2007.

[15] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi, "Tree automata techniques and applications," Available on: http://www.grappa.univ-lille3.fr/tata, 2007, release October, 12th 2007.