Modeling Web Service Composition using Symbolic Transition Systems

Jyotishman Pathak and **Samik Basu** and **Vasant Honavar**
Department of Computer Science
Iowa State University
Ames, IA 50011-1040, USA
{jpathak, sbasu, honavar}@cs.iastate.edu

Abstract

Web services are software entities which provide a set of functionalities that can be accessed over the Web. In general, the valuations of variables appearing in various functions provided by the service are not known before execution. Consequently, to analyze the behavior of a service, all possible valuations need to be considered, making the whole system infinite-state. Against this background, we propose a framework for modeling and composing Web services where desired (goal) and pre-existing (component) services exhibit infinite-state behavior. Our approach finitely represents such services using Symbolic Transition Systems (STSs) which are transition systems augmented with guards over infinite-domain variables. We develop a sound and complete logical approach for identifying the existence of composition of available component-STSs, such that the resulting composition is "simulated" by the goal-STS. In the event that the goal service cannot be realized from the existing components, our technique can also identify the cause for the failure and guide the user to achieve appropriate refinement of the goal specification, thereby paving the way for incremental development of composite services.

Introduction

Recent advances in networks, information and computation grids, and WWW have resulted in the proliferation of a multitude of physically distributed and autonomously developed software components. These developments allow us to rapidly build new and value-added applications from existing ones in various domains such as e-Science, e-Business, and e-Government. However, more often than not, application integration is an engaging and time consuming process because individual software entities are not developed using standard frameworks or component models. Consequently, this results into significant re-designing and re-coupling of existing software leading to substantial loss in productivity. In this context, Service-Oriented Architecture (SOA) (Erl 2004) based on Web services (Alonso *et al.* 2004) are an attractive alternative to build software components and seamless application integration as they offer standardized interface description, discovery and communication mechanisms. More specifically, SOAs provide the ability to construct and deploy complex workflows of composite Web services by leveraging independently developed components.

Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

To explore such opportunities, automatic composition of Web services has emerged as an active area of research in both academia and industry. In a nutshell, automatic service composition amounts to automatically determining a strategy for integrating available component services which would realize the desired/requested service functionality. Generally, there are three ways by which these services can be composed: serial, parallel, or a combination of both serial and parallel. The focus of this paper is to develop an automatic solution for serially composing available Web services into composite services. Even though many approaches (refer to (Dustdar & Schreiner 2005; Hull & Su 2005; Oh, Lee, & Kumara 2005) for a survey) based on AI planning techniques, logic programming, automata-theory etc. have been proposed for doing automatic service composition, they suffer from two major limitations:

- A Web service can provide multiple functionalities which could be invoked over the Web. In general, the valuations of variables appearing in the functions provided by the service are not known before execution. Consequently, to analyze the behavior of a service, all possible valuations need to be considered, making the whole system infinite-state. However, none of the existing approaches for service composition take into consideration the infinite-state behavior typically exhibited by Web services.
- The current techniques provide a "single-step request-response" paradigm for automatic Web service composition. In other words, a user submits a goal specification to a 'composition analyzer', which tries to find an appropriate strategy for realizing the composite service. In the event, such a composition is unrealizable, the whole process fails. As opposed to this, we claim that, there should be provision for iteratively refining the goal specification in an intuitive, yet efficient way, to build composite services.

Against this background, to enhance automation and to minimize the user effort required to develop complex services, we propose *MoSCoE*—a framework for *Modeling Service Composition and Execution* (Figure 1). At its core, MoSCoE is based on the three steps of *abstraction, composition* and *refinement*. The users in our system provide an

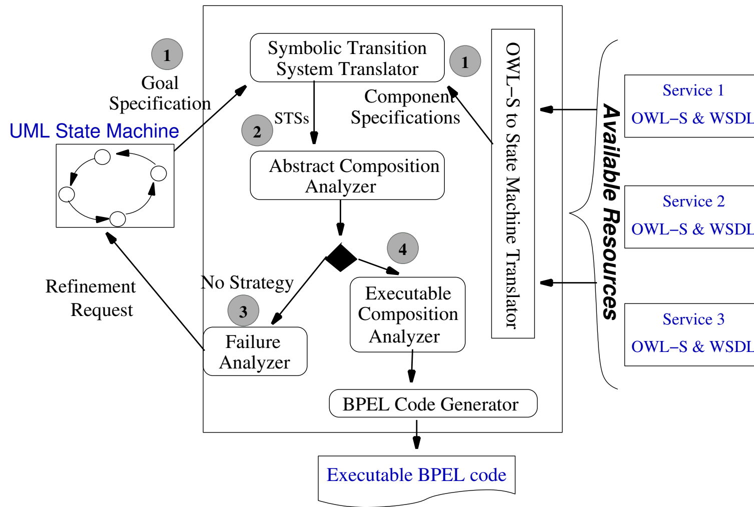

Figure 1: MoSCoE Architectural Diagram. ①: Service requesters and providers present state machine description of the desired/goal and available/component services, respectively. ②: These descriptions are translated into symbolic transition systems which are given as input to the abstract composition analyzer. The analyzer tries to determine a composition strategy. ③: In the event a strategy cannot be computed, the failure analyzer sub-module is invoked which determines the failure-cause information. Appropriately, the user is notified to refine the goal specification. ④: Once a composition strategy is realized, it is translated into a BPEL process flow for execution.

abstract specification of the goal g. Abstraction, in this context, implies that functional description of g is incomplete as the user may not be aware of the details of the available service functions a priori. MoSCoE uses this abstract description and identifies a set of available services, such that their sequential composition realizes (parts-of) g. In the event that such a composition is not realizable, MoSCoE analyzes the cause of the failure and guides the user for appropriate refinement of the goal specification. This process could be done iteratively until a feasible composition is identified or the user decides to abort.

To realize the above functionality, MoSCoE uses UML state machines (Crane & Dingel 2005; Harel 1987) for representing the sequence of functions requested and provided by the goal and component services[1], respectively. However, as noted earlier, Web services typically exhibit infinite-state behavior. To finitely represent the infinite-state systems such as Web services, we rely on using *Symbolic Transition Systems* (STS) (Basu *et al.* 2001) in MoSCoE. An STS can be viewed as a Labeled Transition System (Hopcroft & Ullman 1979) augmented with state variables, guards and functions on transitions. Thus, given a set of available services and a goal service provided by the user, MoSCoE first translates the state machines into their corresponding STS-representation. Once this translation is done, service *composition* in MoSCoE amounts to determining a strategy for sequentially composing the corresponding STS-representations of the components, such that the sequence is "simulation equivalent" to the STS-representation of the goal service. In other words, the sequence of available services *mimics* the functionality of the goal service. However, if such a composition strategy cannot be identified, MoSCoE determines the specific states and transitions in the state machine description of the goal service that needs to be modified, and provides such information to the user for *refinement* purposes. In such a situation, the STS translator is invoked again to generate an STS from the refined goal state machine description.

The contributions of our work are:

- We propose a new paradigm for automatic Web service composition based on abstraction, composition, and refinement. Our approach allows users to iteratively develop composite services from their abstract descriptions.
- The proposed service composition framework takes into consideration the infinite-state behavior typically exhibited by Web services. Such infinite-state behavior is finitely represented using Symbolic Transition Systems (STS).
- In the event that a composition cannot be realized, the proposed approach determines the cause of the failure. Such failure-cause information is essential to realize appropriate refinement of the goal specification and is a vital step for incremental development of complex Web services.

The rest of the paper is organized as follows. We begin by describing the problem of Web service composition, and present a logical formalism for determining feasible composition strategies and failure-cause analysis in infinite-state domain. In the following section, we evaluate our system with a case study to demonstrate the feasibility of our proposed ideas. Finally, we discuss related work in automatic Web service composition, and conclude with future avenues for research.

[1]We assume that the service providers in MoSCoE publish their component services using OWL-S (Martin *et al.* 2004) descriptions. Specifically, the OWL-S "process models" of the component services are translated into corresponding state machines by building on techniques proposed in (Traverso & Pistore 2004).

Service Composition in MoSCoE

Given a set of available component services $s_1, \cdots, s_n$ and a user-specified goal service s_g, service composition synthesis amounts to determining a strategy S for integrating $s_i, \cdots, s_j$, such that S "mimics" s_g. To achieve this, services in our framework are represented as state machines (Crane & Dingel 2005; Harel 1987) which are translated to Symbolic Transition Systems (STSs) (described later in the paper). STSs allow us to finitely represent the infinite-state behavior exhibited by Web services.

Notations

A set of variables, functions and predicates/relations will be denoted by $\mathcal{V}$, $\mathcal{F}$ and $\mathcal{P}$ respectively. The set $\mathcal{B}$ denotes $\{true, false\}$. Elements of $\mathcal{F}$ and $\mathcal{P}$ have pre-defined arities; a function with zero-arity is called a constant. Expressions are denoted by functions and variables, and constraints or guards, denoted by γ, are predicates over other predicates and expressions. Variables in a term t is represented by a set $vars(t)$. Substitutions, denoted by σ, maps variables to expressions. A substitution of variable v to expression e will be represented by $[e/v]$. A term t under the substitution σ is denoted by $t\sigma$.

Service Functions as Transition Systems

A service is described by symbolic transition system where *states* represent current configurations of the service and interstate *transitions* correspond to update to these configurations. Each state is represented by a term, while transitions represent relations between states, and are annotated by *guards, actions* and *effects*. Guards are constraints over term-variables appearing in the `source` state, actions are terms of the form `fname(args)(ret)`; where `fname` is the name of service-function being invoked, `args` is the list of actual arguments and `ret` is the return valuation of the function, if any. Finally, effect is a relation representing the mapping of `source` state variables to `destination` state variables. Formally,

Definition 1 (Symbolic Transition System(Basu *et al.* 2001)) *A symbolic transition system is a tuple $(S, \longrightarrow, s^0, S^F)$ where S is a set of states represented by terms, $s^0 \in S$ is the start state, $S^F \subseteq S$ is the set of final states and $\longrightarrow$ is the set of transition relations where $s \xrightarrow{\gamma,\alpha,\rho} t$ is such that*

1. *γ, $vars(\gamma) \subseteq vars(s)$*
2. *α is a term representing service-functions of the form $a(\vec{x})(y)$ where $\vec{x}$ represents the input parameters and y denotes the return valuations.*
3. *ρ relates $vars(s)$ to $vars(t)$*

Figure 2 shows example STSs. Transitions with no explicit guards are always enabled and function symbols (f, g and h) with no arguments (inputs and outputs) represents constants. Start states do not have any incoming transition and final states are circled.

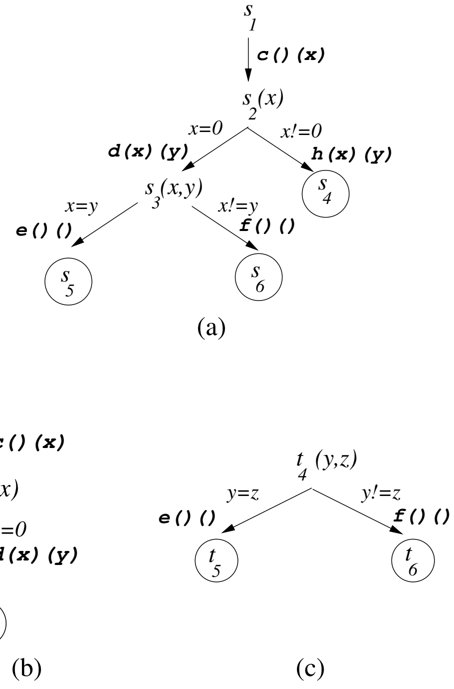

Figure 2: Example Symbolic Transition Systems. (a) $\texttt{STS}_g$ (b) $\texttt{STS}_1$ (c) $\texttt{STS}_2$.

Semantics of STS. The semantics of STS is given with respect to substitutions of variables present in the system. A state represented by the term s is interpreted under substitution σ ($s\sigma$). A transition $s \xrightarrow{\gamma,\alpha,\rho} t$, under *late semantics*, is said to be *enabled* from $s\sigma$ if $\gamma\sigma = \texttt{true}$ and $\gamma \Rightarrow \rho$. The transition under substitution σ is denoted by $s\sigma \xrightarrow{\alpha\sigma} t\sigma$.

Sequential Composition of STS. Sequential composition of $\texttt{STS}_i$ and $\texttt{STS}_j$, denoted by $\texttt{STS}_i \circ \texttt{STS}_j$, is obtained by merging the final states of $\texttt{STS}_i$ with the start state of $\texttt{STS}_j$, i.e., every out-going transition of start state of $\texttt{STS}_j$ is also the out-going transition of each final state of $\texttt{STS}_i$.

In the context of Web services, we say that given a goal service representation $\texttt{STS}_g$ and a set of component representations $\texttt{STS}_i$, the former is said to be realizable from the latter if there exists a sequential composition of components such that $\texttt{STS}_g$ *simulates* $\texttt{STS}_1 \circ \texttt{STS}_2 \circ \ldots \texttt{STS}_n$. In essence, simulation relation ensures that the composition can mimic or replicate (part-of) the goal service functionality. We proceed with the definition of simulation in the context of STSs required to identify a sequential composition as described above.

Definition 2 (Late Simulation) *Given an $STS = (S, \longrightarrow, s^0, S^F)$, late simulation relation with respect to substitution θ, denoted by $\mathcal{R}^\theta$, is a subset of $S \times S$ such that*

$$s_1 \mathcal{R}^\theta s_2 \Rightarrow \quad (\forall s_1\theta \xrightarrow{\alpha_1} t_1\theta . \exists s_2\theta \xrightarrow{\alpha_2} t_2\theta . \; \forall\sigma . \alpha_1\theta\sigma = \alpha_2\theta\sigma \wedge \; t_1 \mathcal{R}^{\theta\sigma} t_2)$$

Two states are equivalent with respect to simulation relation if they are related by the largest similarity relation $\mathcal{R}$.

An $\texttt{STS}_i$ is simulated by $\texttt{STS}_j$, denoted by $(\texttt{STS}_i\, \mathcal{R}^{\theta} \texttt{STS}_j)$ $\texttt{iff}$ $(s_i^0\, \mathcal{R}^{\theta} s_j^0)$.

Example. Consider the example $\texttt{STS}$s in Figure. 2(a) and (b); $\texttt{STS}_1$ is simulated by $\texttt{STS}_g$ according to the following equivalences obtained from above definition.

$$\begin{array}{l} t_1\, \mathcal{R}^{\texttt{true}}\, s_1 \\ \Rightarrow \forall x.(t_2(x)\, \mathcal{R}^{x}\, s_2(x)) \\ \Rightarrow (t_2(x)\, \mathcal{R}^{x=0}\, s_2(x)) \;\wedge\; (t_2(x)\, \mathcal{R}^{x!=0}\, s_2(x)) \\ \Rightarrow (\forall y.(t_3\, \mathcal{R}^{x=0,y} s_3(x,y))) \;\wedge\; \texttt{true} \end{array} \tag{1}$$

In the above example, WLOG we assumed that the variable names in the two $\texttt{STS}$s are identical (x, y) for simplicity.

Proceeding further, we define a termination relation based on late simulation as follows:

Definition 3 (Termination Relation) *Given an* $STS = (S, \longrightarrow, s^0, S^F)$*, termination relation, denoted by* $\mathcal{T}^{\theta,\delta}$ *is a subset of* $S \times S \times S$ *such that*

$$\begin{array}{rl} s_1 \mathcal{T}_t^{\theta,\delta} s_2 \Leftarrow & s_1\, \mathcal{R}^{\theta} s_2 \wedge ((\exists s_1\theta \xrightarrow{\alpha_1} t_1\theta.\ \exists s_2\theta \xrightarrow{\alpha_2} t_2\theta. \\ & \exists \sigma.\alpha_1\theta\sigma = \alpha_2\theta\sigma \wedge\ t_1 \mathcal{T}_t^{\theta\sigma,\delta} t_2) \\ & \vee (s_1 \in S^F \wedge t = s_2 \wedge \delta = \theta)) \end{array}$$

The states s_1 and s_2 are terminally equivalent with respect to t, if they are related by the least solution of terminal relation $\mathcal{T}^{\theta,\delta}$. We say that $(\texttt{STS}_i\ \mathcal{T}_t^{\theta,\delta}\ \texttt{STS}_j)$ $\texttt{iff}$ $(s_i^0\ \mathcal{T}_t^{\theta,\delta}\ s_j^0)$.

Example. Going back to the example in Figure 2(a) and 2(b),

$$\begin{array}{l} t_1\ \mathcal{T}_t^{\texttt{true},\delta}\ s_1 \\ \Leftarrow t_1\ \mathcal{R}^{\texttt{true}}\ s_1 \wedge \exists x.(t_2(x)\ \mathcal{T}_t^{x,\delta}\ s_2(x)) \\ \Leftarrow \texttt{true} \;\wedge\; (\ (t_2(x)\ \mathcal{T}_t^{x=0,\delta}\ s_2(x)) \vee \\ \qquad\quad (t_2(x)\ \mathcal{T}_t^{x!=0,\delta}\ s_2(x))\ ) \\ \Leftarrow (t_2(x)\ \mathcal{R}^{x=0}\ s_2(x) \wedge \exists y.t_3\ \mathcal{T}_t^{\{x=0,y\},\delta} s_3(x,y)) \\ \vee\ \texttt{false} \\ \Leftarrow \texttt{true} \;\wedge\; t = s_3(x,y) \;\wedge\; \delta = \{x=0, y\} \end{array} \tag{2}$$

From the above definitions, it follows that

$$\begin{array}{l} \forall\theta.(\texttt{STS}_1 \circ \texttt{STS}_2 \circ \ldots \circ \texttt{STS}_n)\ \mathcal{R}^{\theta}\ \texttt{STS}_g \equiv \\ T_1 = \{t_1\theta_1 \mid \forall\theta.\exists t_1\theta_1.(\texttt{STS}_1\ \mathcal{T}_{t_1}^{\theta,\theta_1}\ \texttt{STS}_g)\} \wedge \\ \qquad \forall t_1\theta_1 \in T_1.(\texttt{STS}_2 \circ \texttt{STS}_3 \circ \ldots \circ \texttt{STS}_n)\ \mathcal{R}^{\theta_1}\ t_1 \end{array} \tag{3}$$

That is, the composition is realizable via iterative computation of termination relation of the goal transition system against a component. The iterative process terminates when

$$\begin{array}{l} T_n = \{t_n\theta_n \mid \forall t_{n-1}\theta_{n-1} \in T_{n-1}.\exists t_n\theta_n. \\ \qquad (\texttt{STS}_n\ \mathcal{T}_{t_n}^{\theta_{n-1},\theta_n}\ t_{n-1})\} \\ \qquad \wedge\, T = \{t_n \mid t_n\theta_n \in T_n\} \subseteq S_g^F \end{array}$$

Example. In Figure 2, $t_1\ \mathcal{T}_{s_3(x,y)}^{\texttt{true},\{x=0,y\}}\ s_1$ (from Equation 2). Proceeding further and selecting $\texttt{STS}_2$,

$$\begin{array}{rl} & \forall x = 0.\forall y.\exists t\delta.\ t_4(x,y)\ \mathcal{T}_t^{\{x=0,y\},\delta}\ s_3(x,y) \\ \Leftarrow & t_4(x,y)\ \mathcal{T}_t^{\{x=0,x=y\},\delta}\ s_3(x,y)\ \texttt{or} \\ & t_4(x,y)\ \mathcal{T}_t^{\{x=0,x!=y\},\delta}\ s_3(x,y) \\ \Leftarrow & t_4(x,y)\ \mathcal{T}_{s_5}^{\{x=0,x=y\},\{x=0,x=y\}}\ s_3(x,y)\ \texttt{or} \\ & t_4(x,y)\ \mathcal{T}_{s_6}^{\{x=0,x!=y\},\{x=0,x!=y\}}\ s_3(x,y) \end{array}$$

In the above, $\{s_5, s_6\}$ obtained from $\mathcal{T}$ relation is a subset of S_g^F. Therefore, sequential composition, $\texttt{STS}_1$ followed by $\texttt{STS}_2$, is identified which can realize the (part-of) goal $\texttt{STS}_g$.

Failure Analysis. If a composition is not realizable with a specific selection of component-ordering, the above iterative procedure fails. In such scenarios, it is required to identify specific states of the components that cannot be simulated by the corresponding goal states. The motivation is to provide users the cause of the failure in a precise and succinct fashion such that the information can be effectively applied to refine the goal. The failure to simulate a component by single or multiple states in the goal is obtained by defining the *dual* of greatest fixed point simulation relation $\mathcal{R}$.

$$\begin{array}{r} s_1\ \overline{R}^{\theta}\ s_2\ \Leftarrow\ \exists s_1\theta \xrightarrow{\alpha_1} t_1\theta.\forall s_2\theta \xrightarrow{\alpha_2} t_2\theta.\exists\sigma. \\ (\alpha_1\theta\sigma = \alpha_2\theta\sigma) \Rightarrow t_1\ \overline{R}^{\theta\sigma}\ t_2 \end{array} \tag{4}$$

Two states are said be *not* simulation equivalent if they are related by the least solution of $\overline{\mathcal{R}}$. We say that $\texttt{STS}_i\ \overline{\mathcal{R}}^{\theta}\ \texttt{STS}_j$ $\texttt{iff}$ $s_i^0\ \overline{\mathcal{R}}^{\theta}\ s_j^0$.

From Equation 4, the cause of the s_1 *not* simulated by s_2 can be due to

1. $\exists\sigma.\alpha_1\theta\sigma \neq \alpha_2\theta\sigma$, or
2. $\exists\sigma.\alpha_1\theta\sigma = \alpha_2\theta\sigma$ and the subsequent states are related by $\overline{\mathcal{R}}^{\theta\sigma}$, or
3. $\exists s_1\theta \xrightarrow{\alpha_1} t_1\theta$, but there is no transition enabled from s_2 under the substitution θ.

The relation $\overline{\mathcal{R}}$ is, therefore, extended to $\overline{\mathcal{R}}_f$, where f records the *exact* state-pairs which are *not* simulation equivalent.

$$\begin{array}{rl} s_1\ \overline{R}_f^{\theta}\ s_2 \Leftarrow & \exists s_1\theta \xrightarrow{\alpha_1} t_1\theta.(\forall s_2\theta \xrightarrow{\alpha_2} t_2\theta. \\ & \exists\sigma.(\alpha_2\theta\sigma \neq \alpha_1\theta\sigma\ \wedge f = (s_1, s_2, \sigma)) \\ & \vee (\alpha_2\theta\sigma = \alpha_1\theta\sigma\ \wedge t_1\overline{R}_f^{\theta\sigma} t_2)\ ) \\ & \vee (\ \not\exists s_2\theta \longrightarrow\ \wedge f = (s_1, s_2, \theta)) \end{array} \tag{5}$$

Example. Consider the $\texttt{STS}$s in Figure 3. The component $\texttt{STS}$ is not simulated by the first $\texttt{STS}_g$ (rooted at s_1) as there exists a transition from $t_2(x)$ to t_4 when the x is not equal to zero, which is absent from the corresponding state $s_2(x)$ in the goal.

$$\begin{array}{l} t_1 \overline{\mathcal{R}}_f^{\texttt{true}} s_1 \\ \Leftarrow \exists x.t_2(x)\ \overline{\mathcal{R}}_f^{x}\ s_2(x) \\ \Leftarrow (t_2(x)\ \overline{\mathcal{R}}_f^{x!=0}\ s_2(x))\ \texttt{or}\ (t_2(x)\ \overline{\mathcal{R}}_f^{x=0}\ s_2(x)) \\ \Leftarrow (t_3\ \overline{\mathcal{R}}_f^{x!=0}\ s_3)\ \texttt{or}\ (t_2(x)\ \overline{\mathcal{R}}_{(t_2(x),s_2(x),x!=0)}^{x=0}\ s_2(x)) \\ \Leftarrow \texttt{false}\ \texttt{or}\ (t_2(x)\ \overline{\mathcal{R}}_{(t_2(x),s_2(x),x!=0)}^{x=0}\ s_2(x)) \end{array}$$

The state t_1 is also not simulated by goal state s_4 in Figure 3(b) as the state $t_2(x)$ is not simulated by $s_5(y)$; the reason being x and y may not be unified as the former is generated from the output of a transition while the latter is generated at the state. In fact, a state which generates a variable is not simulated by any state if there is a guard on the

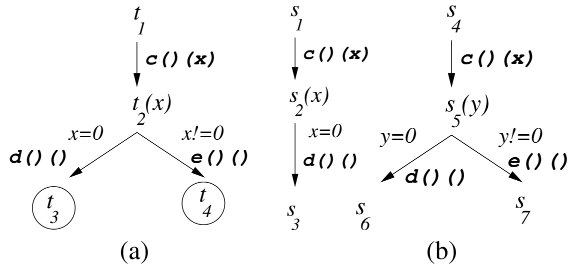

Figure 3: (a) Component STS. (b) Goal STSs $\mathtt{STS}_g$

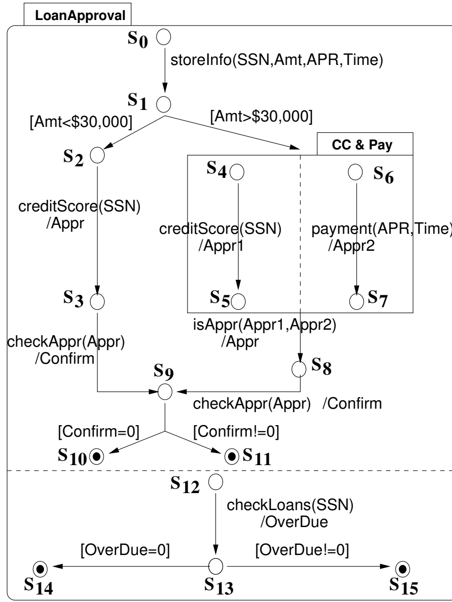

Figure 4: State machine representation of LoanApproval

generated variable. The variables generated at states are *local* to that transition system and cannot be mimicked by another transition system. As such in the example, $t_2(x)$ is not simulated by $s_5(y)$.

Implementation

In (Basu *et al.* 2001), we developed a local and symbolic (bi)-simulation checker for STS using tabled-constraint logic programming in XSB (Sagonas, Swift, & Warren 1993). We apply the same formalism to identify composition of services where services are represented using STSs. In short, composition can be effectively encoded as a tabled logic program, the least model solution of which identifies (negation of) simulation between component STSs. Constraint-handling is essential in our context as STSs are guarded transition systems where guards are defined over constraints on infinite-domain variables. More details about our implementation are available at `http://www.cs.iastate.edu/~jpathak/moscoe.html`.

Case Study: Bank Loan Approval

We present a simple example where a user is assigned to develop a new Web service, `LoanApproval`, which allows potential clients to determine whether an amount of loan requested will be approved or not. The service takes as input the loan amount and social security number (SSN) of the client along with the annual percentage rate (APR) and payment duration (in months) of the loan. Additionally, to assist in the decision-making process, the service also checks payment overdues of the client for his/her existing loans (if any). The user models the service using UML state-machine (Figure 4), where events represent the functions along with their arguments, and effects (pre-pended with "/") represent the outputs of the corresponding function. Guards on transitions are enclosed in "[...]". The user modularizes the design of the desired service using composite states in the state machines; e.g. `CC & Pay` is present inside `LoanApproval`, and there are "and-" states where each partition is separated by dotted lines. A composite "and-state" is one where transitions in different partitions can interleave in any order. Furthermore, the system exits an "and-state" only when all the partitions are in their respective final states. In Figure 4, the final states are represented by solid circles. The corresponding transition system[2] of `LoanApproval` is presented in Figure 5. Transitions with no function invocation makes a call to a dummy function `null` and the dotted lines represent sequence of transitions (not shown) originating due transitions from the and-partition (s_{12} in this case).

Two pre-existing/component services, `Approver` (Figure 6(a)) and `Checker` (Figure 6(b)), are also provided. We assume that, the OWL-S descriptions of these component services are available to our framework. We further assume that the component service descriptions (in particular, OWL-S Process Models) are translated to symbolic transition systems using techniques similar to those described in (Traverso & Pistore 2004).

To determine whether `LoanApproval` can be realized using `Approver` and `Checker` services, $\mathtt{STS}_{\mathtt{LA}}$ simulates a sequential composition of $\mathtt{STS}_{\mathtt{App}}$ and $\mathtt{STS}_{\mathtt{Chck}}$. If the component `Approver` is selected first, $\mathtt{STS}_{\mathtt{App}}$ is *late*-simulated by $\mathtt{STS}_{\mathtt{LA}}$. The paths starting from $s_{0,12}$ in `LoanApproval` simulates the paths in `Approver` such that $s_{10,12}$ is the terminal state corresponding to t_9 and $s_{11,12}$ is the terminal state corresponding to t_{10} (see Figures 5 and 6(a)). In other words, $t_0\ \mathcal{T}^{\mathtt{true},Confirm=0}_{s_{10,12}}\ s_{0,12}$ and $t_0\ \mathcal{T}^{\mathtt{true},Confirm!=0}_{s_{11,12}}\ s_{0,12}$. Proceeding further, $\mathtt{STS}_{\mathtt{Chck}}$ is simulated by $s_{10,12}$ under the substitution $Confirm = 0$ and by $s_{11,12}$ under the substitution $Confirm! = 0$. Note that, there is another solution to the above problem where `Checker` service is followed by `Approval` service.

Next, consider a different loan checker service `NewChecker` (Figure 7) which functions exactly like the `Checker` service for determining client payment overdues, but additionally checks the criminal record of the client. The service first checks whether the client has

[2] Due to brevity, we show only a partial view of the complete transition system.

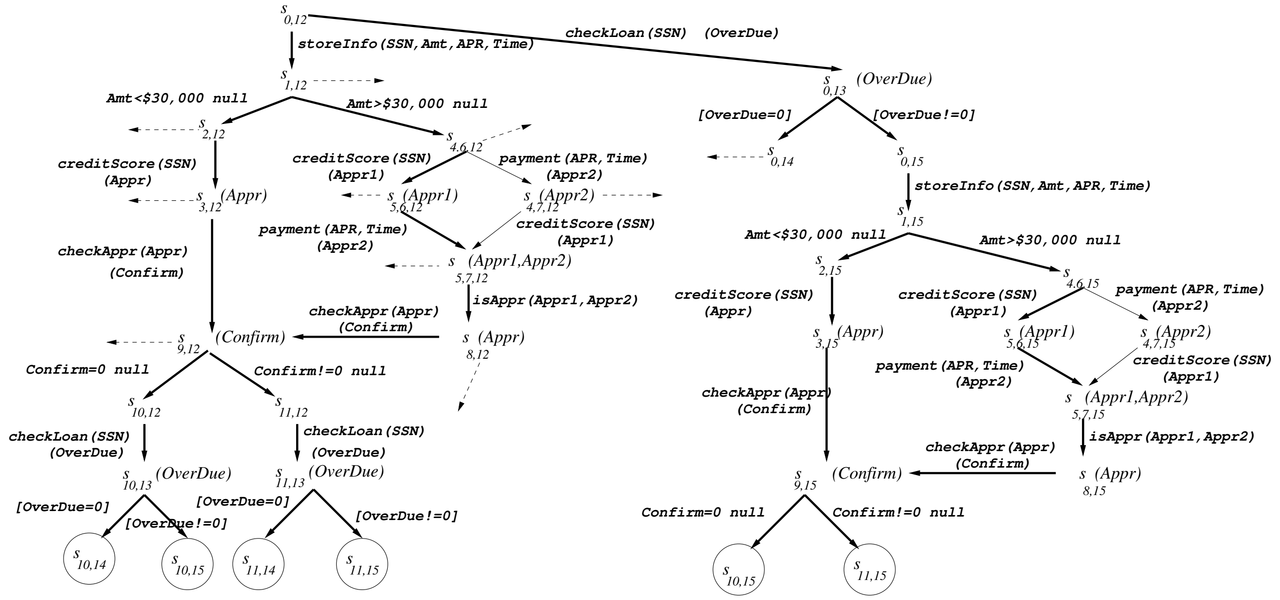

Figure 5: Partial view of `LoanApproval` transition system

payment overdues for the existing loans (if any) and then determines if the client has been previously charged for a criminal act. Since, the 'additional' criminal verification transition is not present in $\mathtt{STS}_{\mathrm{LA}}$,, the component start state t_{11} is not simulated by states $s_{10,12}$, $s_{11,12}$ or $s_{0,12}$, and the transition and substitution causing the simulation-failure can be obtained using $\overline{\mathcal{R}}_f^{\theta}$ relation (see Equation 5). For example, t_{11} is related to $s_{10,12}$ via $\overline{\mathcal{R}}^{OverDue}_{\{t_{12}(OverDue),s_{10,13}(OverDue)\}}$.

Related Work

A number of approaches have been proposed which adopt an transition-system based framework to service composition. Bultan et al. (Bultan *et al.* 2003) model Web services as Mealy machines (finite state machines with inputs and outputs) which are extended with a queue, and communicate by exchanging sequence of asynchronous messages of certain types (called conversations) according to a predefined set of channels. They show how to determine a synthesis of Mealy machines whose conversations, for a given set of channels, satisfy a given specification. This approach is extended in (Fu, Bultan, & Su 2004) where services specified in BPEL (Andrews, Curbera, & et al. 2003) are translated into guarded automata with local XML variables to handle rich data manipulation. The Colombo framework (Berardi *et al.* 2005) is a further extension of the approach proposed in (Bultan *et al.* 2003; Fu, Bultan, & Su 2004), which deals with infinite data values and atomic processes. Colombo models services as labeled transition systems and define composition semantics via message passing, where the problem of determining a feasible composition is reduced to satisfiability of a deterministic propositional dynamic logic formula. Pistore et al. (Pistore *et al.* 2005; Traverso & Pistore 2004) represent Web services using nondeterministic state transition systems, which also communicate through messaging. Their approach however, relies on symbolic model checking techniques to determine a composition strategy. In contrast, services in our framework are represented using Symbolic Transition Systems (STSs) (Basu *et al.* 2001) which are transition systems augmented with guards over infinite-domain variables. STSs allow us to represent infinite-state behavior, which is normally exhibited by Web services. Furthermore, we apply *late*-operational semantics of STS to identify feasible composition strategies.

A few approaches have also been developed which rely on logic programming techniques for doing (semi-) automatic service composition. Rao et al. (Rao, Kungas, & Matskin 2004) translated semantic Web service descriptions in DAML-S (predecessor of OWL-S (Martin *et al.* 2004)) to extralogical axioms and proofs in linear logic, and used π-calculus for representing and determining composite services by theorem proving. Waldinger (Waldinger 2001) also illustrated an approach for service composition using the SNARK theorem prover. This approach is based on automated deduction and program synthesis, where available services and user requirements are described in a first-order language, from which constructive proofs are generated for extracting service composition descriptions. Lämmermann (Lämmermann 2002) proposed using Structural Synthesis of Program (SSP) for automated service composition. SSP uses propositional variables as identifiers for input/output parameters of service functions and applies intuitionistic propositional logic for solving the composition problem. McIlraith and Son (McIlraith & Son 2002) adapt and extend Golog logic programming language, which is built on top of situation calculus, for automatic construction of composite Web services. The approach allows requesters to spec-

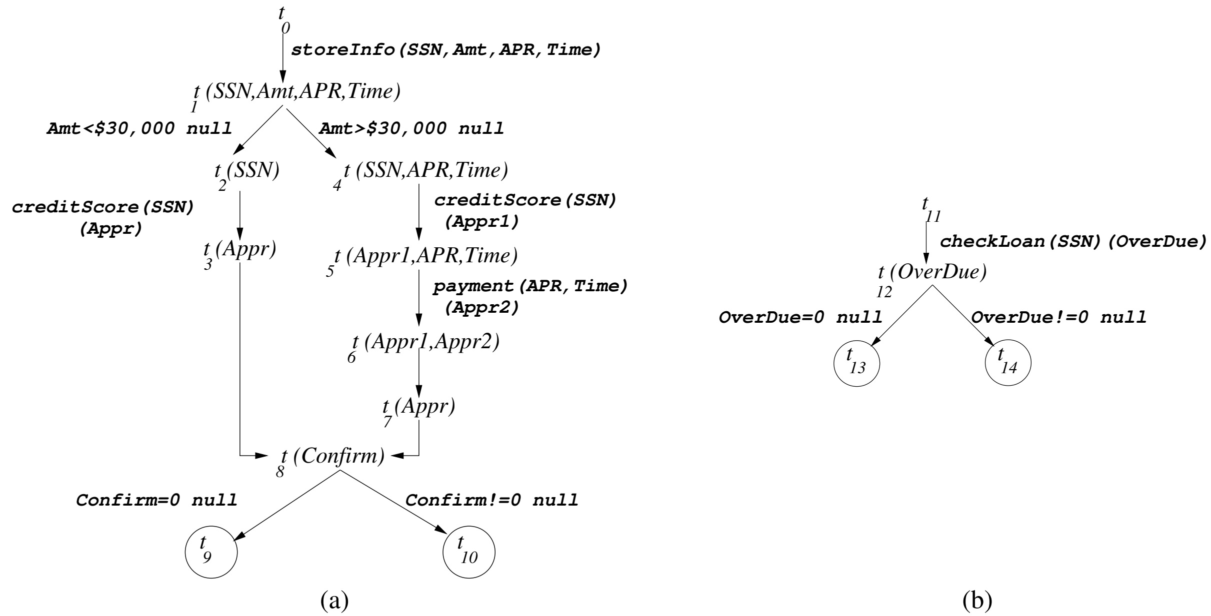

Figure 6: (a) STS representation of `Approver`. (b) STS representation of `Checker`.

ify goal using high-level generic procedures and customizable constraints, and adopts Golog as a reasoning formalism to satisfy the goal. Our approach, on the other hand, uses XSB tabled-logic programming environment (Sagonas, Swift, & Warren 1993) for encoding of the composition algorithm. Tabling in XSB ensures that our composition algorithm will terminate in finite steps as well as avoid repeated sub-computations.

Furthermore, all the efforts mentioned above suffer from one major limitation in that they adopt a "single-step request-response" paradigm for service composition. In other words, if the goal specification provided by the user to the composition analyzer cannot be realized using the available components, the whole process fails. In contrast, our framework allows the users to iteratively refine the goal in an intuitive, yet efficient way, to build composite services. In addition, the existing approaches requires expert knowledge of various description languages, such as OWL-S, BPEL or even more complicated labeled transition systems, to describe and develop composite Web services correctly. Instead, we want to reduce the user burden for learning these technologies, and adopt an abstract model-driven approach for specification of composite services using graphical models such as UML state machines which are easy to understand. Although there have been a few efforts in this direction (Pfadenhauer, Dustdar, & Kittl 2005), they focus mostly on dynamically selecting and binding components as opposed to automatically generating a service composition strategy.

Summary and Discussion

We introduce a novel approach for doing automatic composition of Web services. The framework adopts a symbolic transition system-based approach for computing feasible composition strategies. This formalism allows us to represent finitely the infinite-state behavior of Web services. Our technique provides a goal-directed approach for Web service composition, i.e., we explore only those states/transitions of the goal and component services which could potentially be part of a composition strategy. Furthermore, instead of following the traditional "single-step request-response" approach for service composition, our framework provides users with failure-cause notification for refining the goal specification such that it can be realized by the components. This paves the way to incremental development of complex Web services.

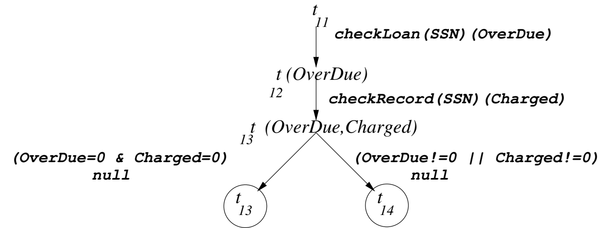

Figure 7: STS representation of `NewChecker`

An important future avenue of research is to investigate situations where components can be composed in parallel (instead of sequential composition as described in this paper). It should be noted the if components need to be composed in a parallel fashion, our solution can be formulated in exactly the same way. Furthermore, we will not be required to generate any strategy as all the component services can be invoked simultaneously. However, the problem is much more involved and requires further study if the components

need to be composed *both* in parallel and sequential fashion. In terms of implementation, we plan to work on automatic translation of the composition strategy into BPEL process flow code, which could be executed to realize the composite service. We also intend to develop heuristics for hierarchically arranging failure-causes to reduce the number of refinement steps typically performed by the user to realize a feasible composition. Finally, we would like to explore ontology-based matchmaking approaches (Pathak *et al.* 2005) to select component services for composition based on their non-functional aspects.

Acknowledgment. This research has been supported in parts by the Iowa State University Center for Computational Intelligence Learning and Discovery (`http://www.cild.iastate.edu`), NSF-ITR grant 0219699 to Vasant Honavar, and NSF grant 0509340 to Samik Basu. The authors would like to thank Robyn Lutz for valuable discussions in early phases of this work and also for her help in preparing this manuscript.

References

Alonso, G.; Casati, F.; Kuna, H.; and Machiraju, V. 2004. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag.

Andrews, T.; Curbera, F.; and et al. 2003. Business Process Execution Language for Web Services, Version 1.1. In *http://www.ibm.com/developerworks/library/ws-bpel/*.

Basu, S.; Mukund, M.; Ramakrishnan, C. R.; Ramakrishnan, I. V.; and Verma, R. M. 2001. Local and Symbolic Bisimulation Using Tabled Constraint Logic Programming. In *Intl. Conference on Logic Programming*, volume 2237, 166–180. Springer-Verlag.

Berardi, D.; Calvanese, D.; Giuseppe, D. G.; Hull, R.; and Mecella, M. 2005. Automatic Composition of Transition-based Semantic Web Services with Messaging. In *31st Intl. Conference on Very Large Databases*, 613–624.

Bultan, T.; Fu, X.; Hull, R.; and Su, J. 2003. Conversation Specification: A New Approach to Design and Analysis of e-Service Composition. In *12th Intl. Conference on World Wide Web*, 403–410. ACM Press.

Crane, M., and Dingel, J. 2005. On the Semantics of UML State Machines: Categorization and Comparision. In *Technical Report 2005-501, School of Computing, Queen's University, Canada*.

Dustdar, S., and Schreiner, W. 2005. A Survey on Web Services Composition. *International Journal on Web and Grid Services* 1(1):1–30.

Erl, T. 2004. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall, New Jersey.

Fu, X.; Bultan, T.; and Su, J. 2004. Analysis of Interacting BPEL Web Services. In *13th Intl. conference on World Wide Web*, 621–630. ACM Press.

Harel, D. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8(3):231–274.

Hopcroft, J. E., and Ullman, J. D. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Welsey.

Hull, R., and Su, J. 2005. Tools for Composite Web Services: A Short Overview. *SIGMOD Record* 34(2):86–95.

Lämmermann, S. 2002. *Runtime Service Composition via Logic-Based Program Synthesis*. Ph.D. Dissertation, Department of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, Sweden.

Martin, D.; Burstein, M.; Hobbs, J.; and et al. 2004. OWL-S: Semantic Markup for Web Services, Version 1.1. In *http://www.daml.org/services/owl-s*.

McIlraith, S., and Son, T. 2002. Adapting Golog for Composition of Semantic Web Services. In *8th Intl. Conference on Principles of Knowledge Representation and Reasoning*, 482–493.

Oh, S.-C.; Lee, D.; and Kumara, S. 2005. A Comparative Illustration of AI Planning-based Web Services Composition. *ACM SIGecom Exchanges* 5(5):1–10.

Pathak, J.; Koul, N.; Caragea, D.; and Honavar, V. 2005. A Framework for Semantic Web Services Discovery. In *7th ACM Intl. Workshop on Web Information and Data Management*, 45–50. ACM press.

Pfadenhauer, K.; Dustdar, S.; and Kittl, B. 2005. Challenges and Solutions for Model Driven Web Service Composition. In *14th IEEE Intl. Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises*, 126–131. IEEE Press.

Pistore, M.; Traverso, P.; Bertoli, P.; and Marconi, A. 2005. Automated Synthesis of Composite BPEL4WS Web Services. In *3rd Intl. Conference on Web Services*, 293–301. IEEE Press.

Rao, J.; Kungas, P.; and Matskin, M. 2004. Logic-based Web Services Composition: From Service Description to Process Model. In *2nd Intl. Conference on Web Services*, 446–453.

Sagonas, K. F.; Swift, T.; and Warren, D. S. 1993. The XSB Programming System. In *Workshop on Programming with Logic Databases, http://xsb.sourceforge.net*.

Traverso, P., and Pistore, M. 2004. Automated Composition of Semantic Web Services into Executable Processes. In *3rd Intl. Semantic Web Conference*, 380–394. Springer-Verlag.

Waldinger, R. J. 2001. Web Agents Cooperating Deductively. In *1st Intl. Workshop on Formal Approaches to Agent-Based Systems*, 250–262. Springer-Verlag.