Selecting and Composing Web Services through Iterative Reformulation of Functional Specifications

Jyotishman Pathak[1,2] **Samik Basu**[1] **Robyn Lutz**[1,3] **Vasant Honavar**[1,2]
[1]Department of Computer Science, Iowa State University, Ames IA 50011
[2]Center for Computational Intelligence, Learning & Discovery, Iowa State University, Ames IA 50011
[3]Jet Propulsion Lab/California Institute of Technology, Pasadena CA 91109
{jpathak, sbasu, rlutz, honavar}@cs.iastate.edu

Abstract

We propose a specification-driven approach to Web service composition. The proposed framework allows users to start with a high-level, possibly incomplete specification of a desired (goal) service that is to be realized using a subset of the available component services. These services are represented by the system using transition systems augmented with guards over variables with infinite domains and are used to determine a strategy for their composition that would realize the goal service. In the event that the goal service cannot be realized using the available services, the system identifies the cause(s) for such failure which can then be used by the developer to reformulate the goal specification. Thus, the system supports Web service composition through iterative refinement of the functional specifications. We present a prototype implementation in tabled-logic programming environment that illustrates the key features of the proposed approach.

1 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 and services in various domains including e-Business and e-Science. Real world applications in these domains call for effective tools for developing composite services using available sets of component services. Consequently, several approaches [6, 8] for doing automatic composition of Web services, including those based on AI planning techniques, logic programming, and automata-theory have been proposed. However, the current approaches have some important limitations:

Reliance on a complete functional specification of the desired service: Existing techniques to service composition [1, 3, 16, 19, 20] require the developer to provide a specification of the desired behavior of the composite service (goal) in its entirety. This forces the developer to work with a complete composition graph. The complexity of such a composition graph (and hence the cognitive burden on the developer) grows rapidly with the increasing complexity of the goal service. In practice, it is difficult to know in advance whether a desired goal service can in fact be realized using the existing components. In many cases, the failure to realize a goal service using a set of component services may be due to incompleteness of the goal specification. In such a setting, it is desirable that the system allows the developer to start with an abstract, and perhaps incomplete specification of the goal service and in the event that the goal service is not realizable using the existing components, guide the developer in reformulating the goal specification so as to reduce the 'gap' between the desired functionality and the capabilities of the existing components.

Inability to handle infinite-state behavior of services: Often, Web services have to cope with a-priori unknown and potentially unbounded data domains (e.g., data types defined by users in WSDL documents). Analyzing the behavior of such a service requires consideration of all possible valuations, which makes the resulting system infinite-state. However, most of the existing approaches to service composition do not take into account the infinite-state behavior exhibited by Web services.

Against this background, we propose *MoSCoE* [13]—a framework for Modeling Service Composition and Execution, based on an approach to service composition through an iterative refinement of the functional specification of the goal service. MoSCoE accepts from the user, an *abstract* (high-level and possibly incomplete) specification of a goal service. In our current implementation, the goal service specification takes the form of an UML state machine that provides a formal, yet intuitive specification of the desired goal functionality. This goal service and the available component services are 'finitely' represented using labeled transition systems augmented with state variables, guards and functions on transitions, namely, Symbolic Transition Systems (STS). Thus, the task of the system is to *compose* a subset of the available component ser-

vices ($c_1, c_2 \cdots c_n$) with the corresponding STS representations ($\texttt{STS}_1, \cdots, \texttt{STS}_n$), such that the result is "simulation equivalent" to the STS-representation ($\texttt{STS}_g$) of the desired goal service g. As noted above, this process might fail either because the desired service cannot be realized using the available component services, or because the specification of the goal service is incomplete. A novel feature of MoSCoE is its ability to identify, in the event of failure to realize a goal service arising from an incomplete goal specification, the specific states and transitions in the state machine description of the goal service that need to be modified. This information allows the developer to *reformulate* the goal specification, and this process[1] can be repeated until a feasible composition is realized, or the user decides to abort.

The contributions of our work include:

- A new paradigm for modeling Web services based on abstraction, composition, and reformulation. The proposed approach allows users to iteratively develop composite services from their abstract descriptions.
- A formalization of the problem of determining feasible compositions in a setting where component services are ordered in a particular fashion and their data (and process) flow is modeled in an infinite-domain.
- An approach to determining the cause of failure of composition to assist the user in modifying and refining the goal specification in an iterative fashion.

The rest of the paper is organized as follows: Section 2 introduces an illustrative example used to explain the salient aspects our work, Sections 3 and 4 present a logical formalism for determining feasible composition strategies and failure-cause analysis in infinite-state domain, respectively; Section 5 discusses the implementation of our composition framework in tabled-logic programming environment [18] and presents a preliminary evaluation, Section 6 briefly discusses related work, and finally Section 7 concludes with future avenues for research.

2 Illustrative Example

We present a simple example where a service developer is assigned to model a new Web service, `KogoBuy`[2], which allows potential clients to order and buy a book online. To achieve this, `KogoBuy` combines two existing and independently developed services: `BookPurchase` and `e-Postal`. `BookPurchase` accepts information about the book (*book name, quantity*) and credit card (*CCNum, CCExpiryDate*) from the client for successful fulfillment of an order. The criteria for a successful order fulfillment is based on two coniderations: (a) the required book and the quantity should be available, and (b) the credit card to be charged should be valid. `e-Postal`, on the other hand, accepts information about the shipping address and the item to be shipped, and determines if the item can be delivered at the particular address. Note that the intention to build `KogoBuy` is to allow the client to interact with `KogoBuy` directly as opposed to interacting with the individual components for buying a book and shipping it to a particular destination.

To model such a scenario in MoSCoE, the user (service developer) needs to provide a goal service specification (e.g., `KogoBuy`) using state machines (Figure 1) consisting of various states and transitions between the states (see Section 3.1). This goal state machine as well as the set of component services are represented internally in MoSCoE using Symbolic Transition Systems (STS, see Section 3.2), which are used to determine a feasible composition strategy that provides the desired functionality. Figure 2 shows the partial transition system of `KogoBuy`, and Figures 3(a) & 3(b) show the transition system of `BookPurchase` and `e-Postal` services, respectively.

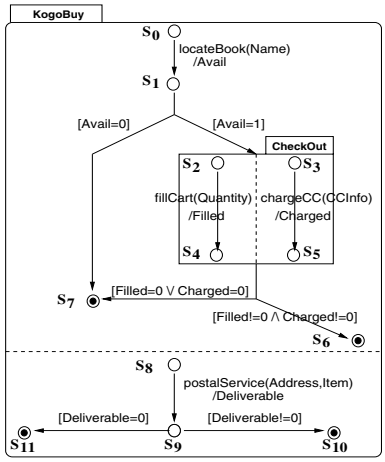

Figure 1: State Machine representation of `KogoBuy`

In practice, there are multiple ways to realize such a composition. We focus on a setting where a service, once invoked, cannot be pre-empted or blocked. This assumption corresponds to scenarios in which services are autonomous entities, and hence their execution behavior cannot be fully controlled by a client. For example, consider a credit card validation service `CCV` which verifies the authenticity of a credit card. Typically, once the client provides the relevant credit card information and initiates the transaction, the client cannot control the execution behavior of the service (other than terminating the execution). Furthermore, our framework disregards those services for composition

[1] Note that determination of the cause for failure of composition, and using that information for reformulation of the goal specification is carried out at *design-time* as opposed to *run-time*.

[2] A variation of the `CongoBuy` composite service presented in http://www.daml.org/services/owl-s.

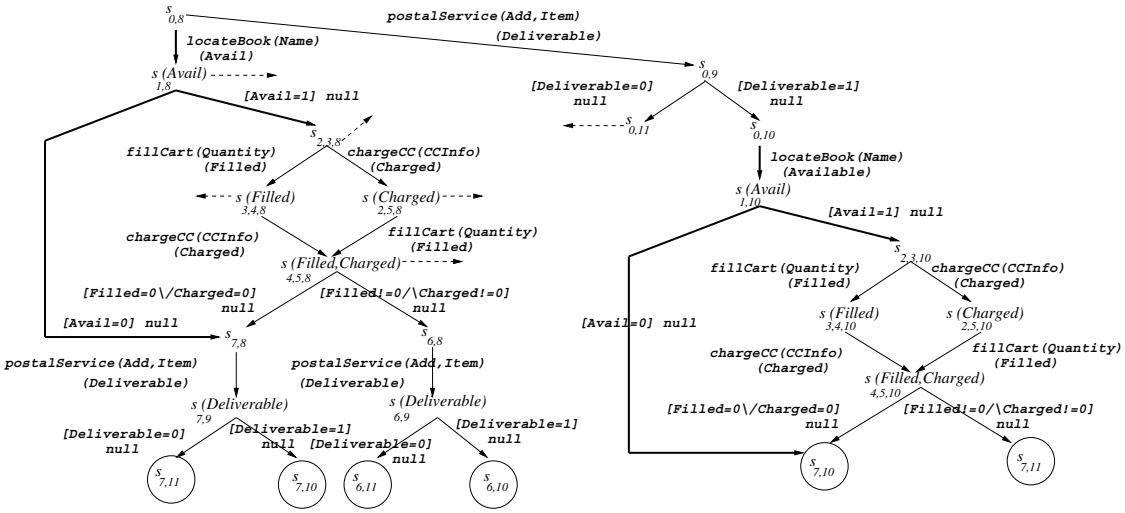

Figure 2: Partial view of KogoBuy transition system

which can can result in behaviors that are beyond those required for achieving the specified goal functionality. For instance, suppose the credit card validation service CCV′ charges a service fee in addition to authenticating the credit card. For a client whose goal is to only validate the authenticity of a card, the additional action is unwarranted. Consequently, we ignore services such as CCV′ for determining a composition strategy from the available pool of services. Note that this approach is in stark contrast to the traditional mediator-based techniques where actions and behaviors that are uncalled-for (as part of the goal) have to be blocked/ignored. Instead, our aim is to find a suitable *ordering* of the available services that can satisfy the goal requirements, an approach we refer as *goal-directed service composition*. MoSCoE solves this composition problem using the notion of *similarity* and *simulation equivalence* (Definition 2)—a composition of components is said to realize a specified goal if the latter simulates the composition, i.e., the composition mimics (part-of) the functional behavior of the goal (see Section 3.3).

3 Service Composition in MoSCoE

3.1 Service Functions as State Machines

We start with a goal specification in the form of a state machine (Figure 1(a)) that consist of states ($s_1, s_2, \cdots$) representing abstraction of the system configuration, and inter-state transitions ($s_1 \longrightarrow s_2$) denoting the conditions under which the system evolves from one state to the next. The states can be either *composite* (or-/and- states) or *atomic*.

Each transition, source $\xrightarrow{ev[g]/e}$ destination, is annotated with *action* labels consisting of *event* (ev), *guard* (g), and *effect* (e). In the context of Web services, the events correspond to various functions that a service provides; the guards refer to pre-conditions of those functions;

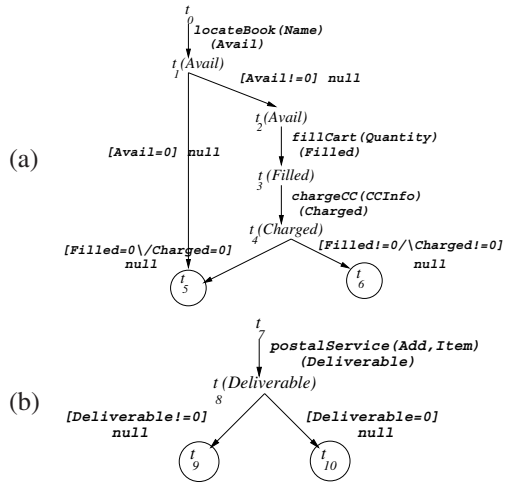

Figure 3: (a) STS representation of BookPurchase. (b) STS representation of e-Postal.

and effects correspond to post-conditions of the transition-functions, in essence denoting the possible assignment of values to variables after the function is executed. A true guard and ϵ (empty) effect denote the absence of pre-/post-conditions, respectively. For example, in Figure 1(a), for transition $s_0 \rightarrow s_1$, the *event* corresponds to function locateBook(Name), the *guard* is assumed to be true, and the *effect* refers to assigning some value to the variable Avail.

Despite their popularity in modeling software, state machines have a major limitation in that they fail to reveal the exact sequence in which the system evolves due to the presence of hierarchy ("and-states") and nesting of sub-states [5]. To address the need to model Web services in such a setting, we use *Symbolic Transition Systems* (STS) [2] to represent Web services (both goal and components) in MoSCoE. In the current context, a state machine can be

translated to the corresponding STS as follows: (a) an STS-state corresponding to an "and-state" is determined by all the active atomic states, (b) an STS-state for an "or-state" corresponds to one of the possible active states, (c) states outside the scope of any "and-/or-composition" are also states in the STS, and finally, (d) initial and end states along with their transitive closures over event-free transitions are start and final states, respectively, of the STS.

3.2 Symbolic Transition System Representation

Preliminaries & Notations. Sets 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 $t\sigma$.

Definition 1 (Symbolic Transition System) *A symbolic transition system is a tuple $(S, \longrightarrow, s0, S^F)$ where S is a set of states represented by terms, $s0 \in S$ 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 is the guard where, $vars(\gamma) \subseteq vars(s)$*
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 $vars(t)$*

Similar to state machines, each state in an STS 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. These states and transitions in STS are annotated by infinite-domain variables and guards over such variables. As such composition using STS (see Section 3.3) requires analysis of all possible (infinite) valuations of STS variables. Figure 4 shows example STSs.

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 $s\sigma \xrightarrow{\alpha\sigma} t\sigma$.

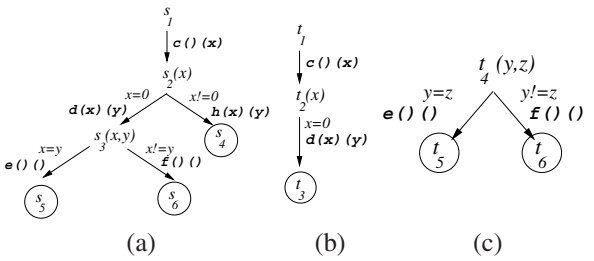

Figure 4: Example Symbolic Transition Systems. (a) $\texttt{STS}_g$ (b) $\texttt{STS}_1$ (c) $\texttt{STS}_2$.

3.3 Composition of Symbolic Transition Systems

A 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 $\texttt{STS}_i$. Recall that our composition does not assume a mediator that can block or ignore unwarranted actions of the component services. As such, composition only considers those components that can provide the actions that are called for by the goal specification. The primary problem, in this case, is to identify the sequence in which the selected components should appear to realize the goal.

We say that given a goal service representation $\texttt{STS}_g$ and a set of component representations $\texttt{STS}_{1\ldots n}$, the former is said to be (partially) realizable from the latter if there exists a composition of components such that $\texttt{STS}_g$ *simulates* $\texttt{STS}_i \circ \texttt{STS}_j \circ \ldots \texttt{STS}_k$. In essence, the simulation relation ensures that the composition can 'mimic' the goal service functionality. We proceed with the definition of simulation in the context of STSs required to identify a feasible composition as described above.

Definition 2 (Late Simulation) *Given an STS $\mathcal{S} = (S, \longrightarrow, s0, S^F)$, 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, under the substitution θ over state variables, are equivalent with respect to simulation if they are related by the *largest* similarity relation $\mathcal{R}^\theta$. We say that an $\texttt{STS}_i$ is simulated by $\texttt{STS}_j$, denoted by $(\texttt{STS}_i \mathcal{R}^\theta \texttt{STS}_j)$ `iff` $(s0_i \mathcal{R}^\theta s0_j)$.

For example, consider the `STS`s in Figure 4(a) and 4(b). If state t_1 is simulated by s_1, then $t_2(x)$ is simulated by $s_2(x)$ for all possible valuations of x. It can be seen that there are two partitions, $x \neq 0$ and $x = 0$, for the infinite domain of variable x. Therefore, $t_2(x)$ is simulated by

$s_2(x)$ for valuations of x in both these partitions. For $x \neq 0$, there is no enabled transition from $t_2(x)$ and as such, it is simulated by $s_2(x)$[3]. On the other hand, when $x = 0$, t_3 is simulated by $s_3(x, y)$ for all possible valuations of y. The above can be represented using logical expressions as follows:

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

Here, WLOG we assumed that the variable names (x, y) in the two STSs are identical for simplicity. Note that the simulation of the $\texttt{STS}_1$ by $\texttt{STS}_g$ leads the latter to state $s_3(x, y)$ with the constraint $x = 0$. In terms of composition, it can be stated that a selected component (simulated by the goal) drives the goal to some specific states. As a result, the start state of the next component in the composition must be simulated by these goal states. To identify the states in the goal that simulates the final states of the component, we define the termination relation:

Definition 3 (Termination Relation) *Given an STS $\mathcal{S} = (S, \longrightarrow, s0, S^F)$, termination relation, denoted by $\mathcal{T}^{\theta,\delta}$ is a subset of $S \times S \times S$ such that*

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

In the above, t represents the states reached after the simulation, and δ and θ are constraints over variables. 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 $\mathcal{T}^{\theta,\delta}$. In other words, s_1 is simulated by s_2 and the final state reached from s_1 is simulated by state t reachable from s_2 under the constraint δ. We say that $(\texttt{STS}_i \ \mathcal{T}_t^{\theta,\delta} \ \texttt{STS}_j)$ iff $(s0_i \ \mathcal{T}_t^{\theta,\delta} \ s0_j)$.

Returning to the example in Figure 4(a) & 4(b), state t_1 is simulated by s_1 and the simulation drives t_1 to $t_2(x)$ and s_1 to $s_2(x)$ where $t_2(x)$ is simulated by $s_2(x)$ for all possible valuations of x ($x \neq 0$ and $x = 0$). For $x \neq 0$, there is no transition from $t_2(x)$ and also, $t_2(x)$ is not a final state in $\texttt{STS}_1$. On the other hand, for $x = 0$, $t_2(x)$ reaches t_3 and this transition is simulated by transition from $s_2(x)$ to $s_3(x, y)$. As t_3 is final state of $\texttt{STS}_1$, the state $s_3(x, y)$ is identified as the state simulating the final state t_3 under the constraint $x = 0$. Thus,

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

[3]A state with no outgoing transition is simulated by all states.

From the preceding discussion,, it follows that $\forall\theta.(\texttt{STS}_i \circ \ldots \circ \texttt{STS}_j) \ \mathcal{R}^{\theta} \ \texttt{STS}_g$ can be realized by identifying the termination relation $\mathcal{T}_{t_1}^{\theta,\theta_1}$ between $\texttt{STS}_i$ and $\texttt{STS}_g$ (t_1 is the state in $\texttt{STS}_g$ which simulates a final state in $\texttt{STS}_i$ under the constraint θ_1) and ensuring that the rest of the components $\texttt{STS}_k \circ \texttt{STS}_l \circ \ldots \circ \texttt{STS}_j$ is simulated by all t_1 with the corresponding θ_1s. Thus,

$$\begin{array}{l} \forall\theta.(\texttt{STS}_i \circ \ldots \circ \texttt{STS}_j) \ \mathcal{R}^{\theta} \ \texttt{STS}_g \equiv \\ T_1 = \{t_1\theta_1 \mid \forall\theta.\exists t_1\theta_1.(\texttt{STS}_i \ \mathcal{T}_{t_1}^{\theta,\theta_1} \ \texttt{STS}_g)\} \\ \wedge \forall t_1\theta_1 \in T_1.(\texttt{STS}_k \circ \texttt{STS}_l \circ \ldots \circ \texttt{STS}_j) \ \mathcal{R}^{\theta_1} \ t_1 \end{array} \quad (3)$$

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 all the states in the goal that simulate the final state of the component under consideration are final states.

$$\begin{array}{l} T_n = \{t_n\theta_n \mid \forall t_{n-1}\theta_{n-1} \in T_{n-1}.\exists t_n\theta_n. \\ \qquad (\texttt{STS}_n \ \mathcal{T}_{t_n}^{\theta_{n-1},\theta_n} \ t_{n-1})\} \\ \qquad \wedge T = \{t_n \mid t_n\theta_n \in T_n\} \subseteq S_g^F \end{array} \quad (4)$$

In the above, T_n represents the set of state-constraint pairs that simulates the final states of the n-th component. If the states in T_n are subset of the final state-set of the goal, then a feasible composition sequence is realized.

For example, in Figure 4, $t_1 \ \mathcal{T}_{s_3(x,y)}^{\texttt{true},\{x=0,y\}} \ s_1$ (from Equation 2). Proceeding further, we select $\texttt{STS}_2$. The start state of $\texttt{STS}_2$ is simulated by the state $s_3(x, y)$ under the constraint $x = 0$ and the states in $\texttt{STS}_g$ that simulate the final states of $\texttt{STS}_2$ are s_5 and s_6. Using termination relation, the result is obtained as follows:

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

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

3.4 Modeling KogoBuy Composite Service

In this section, we show how to model the KogoBuy composite service introduced in Section 2 using the formalisms described above. Figure 2 shows the (partial) transition system of KogoBuy corresponding to its state machine representation (Figure 1(a)). Here, the dotted lines represent sequence of transitions (not shown) originating due to the transitions from the and-partitions (s_8 in this case). Figures 3(a) & 3(b) show the transition system representation for the component services BookPurchase and e-Postal, respectively. To determine whether KogoBuy

can be realized from BookPurchase and e-Postal services, we need to find out if $\texttt{STS}_{\texttt{KB}}$ simulates the composition of $\texttt{STS}_{\texttt{BP}}$ and $\texttt{STS}_{\texttt{eP}}$. If the component BookPurchase is selected first, it can be seen that $\texttt{STS}_{\texttt{BP}}$ is *late*-simulated by $\texttt{STS}_{\texttt{KB}}$. This is because the transition paths starting from state $s_{0,8}$ in KogoBuy simulate the paths in BookPurchase such that $s_{6,8}$ is the terminal state corresponding to state t_6, and similarly $s_{7,8}$ is the terminal state corresponding to state t_5. In other words, we have $t_0 \ \mathcal{T}_{s_{6,8}}^{\texttt{true},(Filled!=0 \& Charged!=0)} \ s_{0,8}$ and $t_0 \ \mathcal{T}_{s_{7,8}}^{\texttt{true},(Filled=0 \| Charged=0)} \ s_{0,8}$. Now, $\texttt{STS}_{\texttt{eP}}$ is simulated by $s_{6,8}$ under the substitution (*Filled!=0* & *Charged!=0*) and by $s_{7,8}$ under the substitution (*Filled=0* $\|$ *Charged=0*). Note that for this simple example there is another solution to build a composition where the e-Postal service is selected first followed by BookPurchase.

4 Failure-Cause Analysis and Goal Reformulation

The composition of a goal service from available component services using the process outlined above will fail when some aspect of the goal specification cannot be realized using the available component services. When this happens, our approach seeks to provide information to the user concerning the cause of the failure in a form that can be used to reformulate the goal specification. In our framework, the reason for failure to simulate a component by one or more states of the goal STS $\texttt{STS}_g$ is obtained by identifying the *dual* of greatest fixed point simulation relation $\mathcal{R}$:

$$s_1 \ \overline{R}^{\theta} \ s_2 \ \Leftarrow \quad \exists s_1\theta \xrightarrow{\alpha_1} t_1\theta. \forall s_2\theta \xrightarrow{\alpha_2} t_2\theta. \exists\sigma. \\ (\alpha_1\theta\sigma = \alpha_2\theta\sigma) \Rightarrow t_1 \ \overline{R}^{\theta\sigma} \ t_2 \tag{5}$$

Two states are said to be *not* simulation equivalent if they are related by the least solution of $\overline{\mathcal{R}}$. We say that $\texttt{STS}_i \ \overline{\mathcal{R}}^{\theta} \ \texttt{STS}_j$ iff $s_i^0 \ \overline{\mathcal{R}}^{\theta} \ s_j^0$. From Equation 5, the cause of the state s_1 *not* simulated by s_2 can be due to:

1. $\exists\sigma.\alpha_1\theta\sigma \neq \alpha_2\theta\sigma$ (i.e., that actions do not match), 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.

$$s_1 \ \overline{R}_f^{\theta} \ s_2 \Leftarrow \quad \exists s_1\theta \xrightarrow{\alpha_1} t_1\theta.(\forall s_2\theta \xrightarrow{\alpha_2} t_2\theta. \\ \exists\sigma.(\alpha_2\theta\sigma \neq \alpha_1\theta\sigma \ \wedge f = (s_1, s_2, \sigma)) \\ \vee (\alpha_2\theta\sigma = \alpha_1\theta\sigma \ \wedge t_1\overline{R}_f^{\theta\sigma} t_2) \ ) \\ \vee (\not\exists s_2\theta \longrightarrow \ \wedge f = (s_1, s_2, \theta)) \tag{6}$$

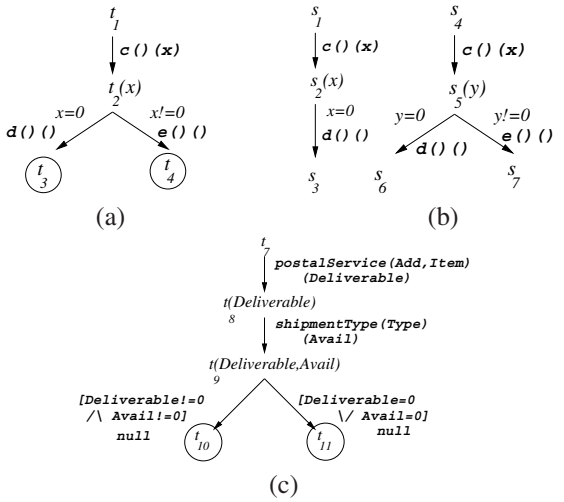

Figure 5: (a) Component STS (b) Goal STSs $\texttt{STS}_g$ & $\texttt{STS}_{g'}$ (c) STS for NewPostal

For example, consider the STSs in Figure 5(a) & 5(b). The component 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. That is,

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

The state t_1 is also not simulated by state s_4 of $\texttt{STS}_{g'}$ as the state $t_2(x)$ is not simulated by $s_5(y)$. This is because 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 generated variable. Such generated variables at the states are *local* to that transition system and hence, cannot be 'mimicked' by another transition system. In our example, $t_2(x)$ is not simulated by $s_5(y)$.

Failure-cause analysis for KogoBuy. Returning to our example from Section 2, assume that we replace the e-Postal component service (Figure 3(b)) with another shipment service NewPostal (Figure 5(c)), which functions exactly like e-Postal, but additionally asks the client to provide information about the shipment type (e.g., overnight, 2^{nd} day air, ground). However, since this 'additional' shipment type transition is not present in $\texttt{STS}_{\texttt{KB}}$, the component start state t_7 is not simulated by states $s_{0,8}$, $s_{6,8}$ or $s_{7,8}$. This information about the transition and substitution causing the failure of simulation can be obtained using the $\overline{\mathcal{R}}_f^{\theta}$ relation (see Equation 6). For example, t_7

is related to $s_{7,8}$ via $\overline{\mathcal{R}}^{Deliverable}_{\{t_8(Deliverable),s_{7,9}(Deliverable)\}}$. Note that this kind of failure-cause information can be provided to the user which can be used for refining the goal specification in an iterative manner. In this case the user can add the shipment transition (with appropriate parameters) to the goal specification and try to determine a feasible composition strategy. These steps can be iterated until a composition strategy is found or the user decides to abort.

5 Prototype Implementation

We have implemented a prototype of MoSCoE in the XSB [18] tabled-logic programming environment. The core of the implementation consists of encoding the simulation relation and the termination relation, and developing a *meta-interpreter*[4] to evaluate the relations in the context of constraints over infinite-domain variables. Our logical encoding is direct and can yield a local, on-the-fly simulation checker, where states and transitions are explored only if they are needed to prove or disprove simulation equivalence. In what follows, we proceed with a brief introduction to XSB (Section 5.1) followed by discussion of encoding of identification of composition of services and generation of failure-cause information if such a composition does not exist. Additional information about the prototype is available at `http://www.cs.iastate.edu/~jpathak/moscoe.html`.

5.1 Preliminaries: XSB Tabled-Logic Programming

XSB logic programming system [18], developed at SUNY Stony Brook, is an extension of Prolog-style SLD resolution with tabling[5]. Tabling enables XSB (a) to terminate with correct results on programs having finite models, (b) to the compute the least model of normal logic programs and (c) to avoid repeated subcomputations. Predicates or relation are defined as rules of the form:

```
G :-  G1, G2, ..., Gn.
```

where the relation `G` evaluates to true if the subgoals `G1` through `Gn` evaluates to true, i.e., `G1` $\wedge$ `G2` $\wedge \ldots \wedge$ `Gn` $\Rightarrow$ `G`. Variables in the subgoals are existentially quantified and rules with no right-hand side of `:-` are referred to as *facts*.

Consider a simple example for computing reachability relation between states of a graph, i.e., transitive closure of edges in the graph. The graph can be defined using logical facts describing the edge relations as follows:

```
edge(s0, s1).
edge(s1, s0).
edge(s2, s1).
```

There are three states in the graph `s0, s1` and `s2`, and there are transitions from (a) `s0` to `s1`, (b) `s1` to `s0` and (c) `s2` to `s1`. The reachability relation can be encoded in XSB as follows:

```
reach(S, T) :- edge(S, T).
reach(S, T) :- edge(S, S1), reach(S1, T).
```

There are two rules that define `reach` predicate or relation. First rule states that `reach(S, T)` is satisfied if there exists an edge between `S` and `T`. The second rule computes the transitive closure stating that `T` can be reached from `S` if `S1` can *reach* `T` and there exists an edge from `S` to `S1`.

Let us consider the evaluation of the query `reach(s0, Ans)` used to find all the states that can be reached from `s0`. According to the first rule, `reach(s0,s0)` and `reach(s0,s1)` evaluate to true. But, application of the second rule, results in an infinite recursion path: `reach(s0, Ans)` depends on the valuation of `reach(s0, Ans)` as there exists an edge from `s0` to `s0`. In other words, the normal evaluation of the logical relation fails to compute the *least model solution* of the above program. The directive `:- table reach/2`, when included in the above program, ensures that the queries and the results of `reach` predicate are memorized and as such self-dependency in a recursive path can be avoided. Thus, if the truth-valuation of `reach(s0, Ans)` depends on the truth-valuation of `reach(s0, Ans)`, the relation `reach(s0, Ans)` evaluates to `false` along this recursive path.

5.2 Logical Encoding of Service Composition

5.2.1 Transition Relations

As outlined, an important element of our implementation is encoding the simulation relation and termination relation, and using a meta-interpreter to evaluate the relations in the context of the constraints over infinite domain variables. The relation `trans` describes the symbolic transition systems as follows:

```
trans(S, G, A, T)
```

where `S, G, A, T` are source state, guard, action and destination states respectively. For example, a transition from $s(x, y) \xrightarrow{x=y, f(x,y)(z)} t(x, z)$ can be encoded as:

```
trans(s(X, Y), X=Y, f([X,Y], Z), t(X, Z)).
```

In the above, the action `f([X,Y],Z)` represents a function `f` with input arguments `X` and `Y`, and output `Z`. Note that we do not include the transfer relation—the mapping of source-state and destination-state variables. This mapping is implicitly understood using unification of logical variables. For example, the transfer relation maps `X` in the source state to that in the destination state.

[4] A meta-interpreter for a language is an interpreter for the language written in the language itself.

[5] Tabling is a technique that can get rid of infinite loops for bounded term-size programs and possible redundant computations in the execution of logic programs. The main idea of tabling is to memorize the answers to some calls and use them to resolve subsequent variant calls.

The semantics of transition relations leads to generation of *late* transition systems as described in Section 3.2.

```
late_trans(S, A, T) :-
   trans(S, Gamma, Alpha, T), Gamma.
```

The semantics require handling of constraints over source-state variables. We use a meta-interpreter to evaluate each transition relation in the context of a set of constraints. The meta-interpreter predicate, `interp(Goal, Cin, Cout)`, takes as argument a `Goal` predicate and evaluates its truth-value in the context of a set of constraints `Cin`. If `Goal` is interpreted to be true in the context of `Cin`, the resultant constraint is `Cout`. For example, `interp(late_trans(S, A, T), Cin, Cout)`, checks whether `Cin` evaluates the guard `Gamma` to true and interprets `A` from `Cin` and `Alpha`.

5.2.2 Dual of Simulation

The relation described in Equation 5 can be encoded as a logic program and interpretation of the least model of the program provides the solution to the corresponding relation.

```
:- table nsim/2.

nsim(S1, S2) :-
   late_trans(S1, A1, T1),
   no_matching_trans(S2, A1, T1).

no_matching_trans(S2, A1, T1) :-
   forall( (A2,T2),
           late_trans(S2, A2, T2),
           nsimulate(A1, T1, A2, T2) ).

nsimulate(A1, T1, A2, T2) :-
   copy_term( (A1, T1, A2, T2),
              (A11, T11, A22, T22) ),
   (
      A11 = A22, nsim(T11, T22)
   ;  A11 \= A22
   ).
```

In the above, meta-interpretation of `nsim`, i.e., `interp(nsim(S1, S2), Cin, Cout)` represents the $\overline{\mathcal{R}}^{\theta}$ relation where `Cin` represents θ. The above program states that, if there exists a transition from `S1` which is not matched by any transition from `S2` (predicate `no_matching_trans`), then `S1` is *not* simulated by `S2`. The predicate `no_matching_trans` aggregates all the transitions from `S2` and invokes `nsimulate` on each element of the aggregation (using predicate `forall`). Hence, in `nsimulate`, we use `copy_term` to produce different copies of action and state variables in `A1,A2` and `T1, T2`. Transitions with actions that produce outputs of functions and variables appearing in the destination states for the first time are *free* variables. Consequently, such variables must be interpreted in different set of constraints for each pair of `A1, A2` and `T1, T2` as they are existentially quantified in Equation 5 (note $\exists\sigma.(\alpha_1\theta_1\sigma = \alpha_2\theta_2\sigma) \Rightarrow t_1\ \overline{R}^{\theta\sigma}\ t_2$). The predicate `nsimulate`, therefore, evaluates to true (a) when `A11=A22` and the subsequent states are not similar; and (b) `A11\=A22`.

The predicate `nsim` can be directly extended to encoding of Equation 6 required to identify the cause of failure of simulation.

5.2.3 Termination Relation

The Definition 3 is encoded using the above encoding of `nsim`. The predicate `terminal(S1, S2, T)` when meta-interpreted in the context of the constraints `Cin` and `Cout` evaluates the solution for $\mathcal{T}_t^{\theta,\delta}$ where `Cin` represents θ, `Cout` represents δ and `T` represents the state t.

```
terminal(S1, S2, S2) :- finalcomp(S1).
terminal(S1, S2, T) :-
  negate nsim(S1, S2),
  late_trans(S1, A1, T1),
  matching_trans(S1, A1, T1, T).

matching_trans(S1, A1, T1, T) :-
  late_trans(S2, A2, T2),
  copy_term( (A1, T1, A2, T2),
             (A11, T11, A22, T22)),
  A11 = A22,
  terminal(T1, T2, T).
```

The first rule `terminal` predicate states that `T` is equal to `S2` if `S1` is the final state of component (base case). Otherwise, check whether the two states are similar (negation of `nsim`), if so identify two matching transitions and recursively invoke `terminal` predicate on the subsequent states. The meta-interpreter `interp` evaluates the negation of `nsim` as follows: if interpretation of `nsim` evaluates to true, the interpretation of its negation evaluates to false, and vice-versa.

5.2.4 Iterative Computation of Composition

Finally, given a set of component-transition relations and a goal transition relation, the composition of the components that can be simulated by the goal is obtained using the predicate `strategy`. In essence, definition of `strategy` is the encoding of Equation 3.

```
strategy(GoalStates, StratIn, StratOut) :-
  startcomponent(S1),
  allterminal(S1, GoalState, Terminals),
  ( subsetoffinalgoal(Terminals)
   ->  Stratout = [S1|StratIn]
   ;   strategy(Terminals, [S1|StratIn],
                        StratOut)
  ).

allterminal(S1, [], []).
allterminal(S1, [S2|S2s], List) :-
    findall(T, terminal(S1, S2, T),
                              List1),
    allterminal(S1, S2s, List2),
    append(List1, List2, List).
```

In the above encoding, a component is selected using `startcomponent` fact. The predicate `allterminal` identifies all the terminal states of the goal `STS` once the component reaches its final states from its start state `S1`. If the terminal states `Terminals` is a subset of the final states of the goal, a composition strategy is obtained which is recorded in `StratOut` by adding `S1` to `StratIn`. Note that the actual composition is the reverse of the list `StratOut`.

5.3 Preliminary Evaluation

We have conducted some preliminary experiments to determine the time taken for identifying: (i) a feasible composition strategy, and (ii) the failure-cause information, if such a strategy cannot be realized. These experiments were based on the `KogoBuy` example illustrated in Section 2. Thus, the goal was to model `KogoBuy` composite service using a set of available services (in this case `BookPurchase` and `e-Postal`). Recall that there are two possible solutions by which such a composition can be realized (see Section 3.4): composing `BookPurchase` followed by `e-Postal`, and composing `e-Postal` followed by `BookPurchase`. We also ran experiments by modifying the available services (obtained by deleting transitions). In all the above scenarios, composition strategy is identified in less than 0.02 seconds. Furthermore, we tried to find a composition between `BookPurchase` and `NewPostal` (see Section 4) and their variations, which resulted in the failure of composition. The failure cause was correctly identified in about 0.01 seconds. As expected, the experiments revealed that time and memory requirements for identifying feasible composition or failure-causes depend on the ordering of the selection of components. We are currently investigating heuristics to obtain component ordering that will lead to efficient time and memory usage. Furthermore, systematic evaluation of the proposed approach on standard benchmarks [12] as well as a real-world application is in progress.

6 Related Work

A number of approaches that adopt a transition-system based framework to service composition have been proposed in the literature. Bultan et al. [4, 7] model Web services as automata extended with a queue, and communicate by exchanging sequence of asynchronous messages, which are used to synthesize a composition for a given specification. Their approach is extended in Colombo [3] 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. [16, 20] represent Web services using non-deterministic 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 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.

Rao et al. [17] translated semantic Web service descriptions in DAML-S to extralogical axioms and proofs in linear logic, and used π-calculus for representing and determining composite services by theorem proving. Waldinger [21] illustrated an approach for service composition using the SNARK theorem prover. The technique is based on automated deduction and program synthesis where constructive proofs are generated for extracting service composition descriptions. McIlraith and Son [10] adapt and extend Golog logic programming language for automatic construction of composite Web services. The approach allows requesters to specify 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 [18] for encoding of the composition framework. Tabling in XSB ensures that our composition algorithm will terminate in finite steps as well as avoid repeated sub-computations, resulting in more efficiency.

Some authors have also focused on using abstract specification for describing goal services by applying model-driven techniques [15]. However, most of this work is geared towards dynamic selection and binding of components, as opposed to generation of composition strategies. In contrast, our emphasis is on the use of abstract specifications (that can be reformulated iteratively) for generating a feasible service composition in an automatic fashion. Additionally, our technique circumvents the necessity of learning Web service specification languages on the part of software engineers who are familiar with UML state machines.

7 Summary and Discussion

We have introduced a novel approach to developing composite services through an iterative reformulation of the goal service specifications. The use of the symbolic transition systems formalism allows us to 'finitely' represent the potentially infinite-state behavior of Web services. Furthermore, in contrast to the traditional approaches for service composition, which require the developer to provide a complete specification of the goal service at the outset, our framework reduces the cognitive burden on the developer by allowing them to begin with an abstract, possibly incomplete specification of the desired goal that can be modified

and refined iteratively so as to 'reduce the gap' between the desired functionality and the capabilities of the available components.

The framework described in this paper is restricted to services which can be specified using a limited class of constraints as guards in the STS. This restriction ensures that the fixed point computation of similarity relation terminates. We plan to investigate a larger class of constraints (e.g., range constraints and arithmetic operations) based on the techniques described in [9]. Work in progress is aimed at developing heuristics for hierarchically arranging failure-causes to reduce the number of refinement steps typically performed by the user to realize a feasible composition and to do a comprehensive evaluation. We also plan to explore ontology-based matchmaking approaches [14] to select component services from semantically heterogeneous component libraries based on their functional and non-functional specifications, and apply techniques for composability checking [11] to avoid failures during execution. Other work in progress is aimed at automatic translation of the composition strategy into BPEL process flow code that can be executed to realize the composite service. Of particular interest to us is a systematic evaluation of scalability and efficiency of the proposed approach on a broad class of benchmark service composition problems [12].

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, NSF grant 0509340 to Samik Basu, and NSF grant 0541163 to Robyn Lutz.

References

[1] V. Agarwal, K. Dasgupta, and et al. A Service Creation Environment Based on End to End Composition of Web Services. In *14th Intl. Conference on World Wide Web*, pages 128–137. ACM Press, 2005.

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

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

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

[5] M. L. Crane and J. Dingel. UML Vs. Classical Vs. Rhapsody Statecharts: Not All Models Are Created Equal. In *8th Intl. Conference on Model Driven Engineering Languages and Systems*, pages 97–112. LNCS 3713, 2005.

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

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

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

[9] R. Kumar, C. Zhou, and S. Basu. Finite Bisimulation of Reactive Untimed Infinite State Systems Modeled as Automata with Variables. In *American Control Conference*, 2006.

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

[11] B. Medjahed and A. Bouguettaya. A Multilevel Composability Model for Semantic Web Services. *IEEE Transactions on Knowledge and Data Engineering*, 17(7):954–968, 2005.

[12] S.-C. Oh, H. Kil, and D. L. and. WSBen: A Web Services Discovery and Composition Benchmark. In *IEEE International Conference on Web Services*, 2006.

[13] J. Pathak, S. Basu, R. Lutz, and V. Honavar. MoSCoE: A Framework for Modeling Web Service Composition and Execution. In *IEEE 22nd Intl. Conference on Data Engineering Ph.D. Workshop*, page x143. IEEE CS Press, 2006.

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

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

[16] M. Pistore, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *15th Intl. Conference on Automated Planning and Scheduling*, pages 2–11, 2005.

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

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

[19] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan. Automated Discovery, Interaction and Composition of Semantic Web Services. *Journal of Web Semantics*, 1(1):27–46, 2003.

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

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