International Journal on Artificial Intelligence Tools
Vol. 17, No. 1 (2008) 109–138
© World Scientific Publishing Company

MoSCoE: AN APPROACH FOR COMPOSING WEB SERVICES THROUGH ITERATIVE REFORMULATION OF FUNCTIONAL SPECIFICATIONS

JYOTISHMAN PATHAK
Division of Biomedical Informatics
Mayo Clinic College of Medicine
Rochester, MN 55905, USA
pathak.jyotishman@mayo.edu

SAMIK BASU
Software Systems Specification & Verification Laboratory
Department of Computer Science, Iowa State University
Ames, IA 50011-1040, USA
sbasu@cs.iastate.edu

ROBYN LUTZ
Laboratory for Software Safety
Department of Computer Science, Iowa State University
Ames, IA 50011-1040, USA
rlutz@cs.iastate.edu

VASANT HONAVAR
Artificial Intelligence Research Laboratory
Department of Computer Science, Iowa State University
Ames, IA 50011-1040, USA
honavar@cs.iastate.edu

We propose a specification-driven approach to Web service composition. Our framework allows the users (or service developers) 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 using labeled 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 functionality. However, in the event the goal service cannot be realized using the available services, our approach identifies the cause(s) for such failure which can then be used by the developer to reformulate the goal specification. Thus, the technique supports Web service composition through iterative reformulation of the functional specification. We present a prototype implementation in a tabled-logic programming environment that illustrates the key features of the proposed approach.

Keywords: Service-oriented architectures; web services; composition; symbolic transition systems; tabled-logic programming.

1. Introduction

Recent advances in networks, information and computation grids, and WWW have resulted in the proliferation of physically distributed and autonomously developed software components. These developments allow us to rapidly build new value-added applications from existing ones in various domains such as e-Science, e-Business, and e-Government. However, often the process of integrating applications becomes tedious and time consuming because individual software entities are not developed using standard frameworks or component models. This results in significant re-designing and re-coupling of existing software leading to substantial loss in productivity.

In this context, Service-Oriented Architectures (SOAs)[13,14] based on Web services[1] that offer standardized interface description, discovery and communication mechanisms are an attractive alternative to build software components and provide seamless application integration. In particular, developing approaches for (semi-) automatic composition of Web services[a] has emerged as an active area of research in both academia and industry. Many recent efforts (see Refs. 12, 18, 25, 27, 41 and 48 for surveys), that leverage techniques based on AI planning, logic programming, and formal methods have focused on different aspects of Web service composition ranging from service discovery to service specification and deployment of composite services. However, despite the progress, the current state of the art in service composition has several limitations:

- *Complexity of Modeling Composite Services:* For specifying functional requirements, the current techniques for service composition require the service developer to provide a specification of the desired behavior of the composite service (goal) in its entirety. Consequently, the developer has to deal with the cognitive burden of handling the entire composition graph (comprising appropriate data and control flows) which becomes hard to manage with the increasing complexity of the goal service. Instead, it will be more practical to allow developers to begin with an abstract, and possibly incomplete, specification that can be incrementally modified and updated until a feasible composition is realized.
- *Inability to Analyze Failure of Composition:* The existing techniques for service composition adopt a 'single-step request-response' paradigm for modeling composite services. That is, if the goal specification provided by the service developer cannot be realized by the composition analyzer (using the set of available component services), the entire process fails. As opposed to this, there is a requirement for developing approaches that will help identify the cause(s) for failure of composition and guide the developer in applying that information for appropriate reformulation of the goal specification in an iterative manner. This requirement is of particular importance in light of the previous limitation because in many

[a]In this paper, we use the terms "service" and "Web service" interchangeably.

cases the failure to realize a goal service using a set of component services can be attributed to incompleteness of the goal specification.

- *Inability to Analyze Infinite-State Behavior of Services:* Often, Web services have to cope with apriori 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 service composition approaches do not take into account the infinite-state behavior exhibited by Web services.
- *Lack of Formal Guarantees:* Formally guaranteeing the soundness and completeness of an algorithm for service composition is a vital requirement for ensuring the correctness of the composite service (generated by the algorithm). In this context, we say that an algorithm is complete if it is able to find a feasible composition whenever it exists, whereas the algorithm is sound if ascertains that the composition generated realizes the goal service. Nevertheless, most of the existing service composition techniques do not provide soundness and completeness guarantees.

Against this background, we propose *MoSCoE*[b],[30,32,34] — an approach for developing composite services in an incremental fashion through iterative reformulation of the functional specification of the goal service. MoSCoE accepts from the user (i.e., service developer), 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[10] that provides a formal, yet intuitive specification of the desired goal functionality. This goal service and the available component services are represented using labeled transition systems augmented with state variables, guards and functions on transitions, namely, Symbolic Transition Systems (STS).[c] Thus, the task of the system is to *compose* a subset of the available component services $(c_1, c_2 \cdots c_n)$ with the corresponding STS representations $(\mathtt{STS}_1, \cdots, \mathtt{STS}_n)$, such that the resultant composition "realizes" the desired goal service $\mathtt{STS}_g$. 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 require modification. This information allows the developer to *reformulate* the goal specification, and the procedure[d] can be repeated until a feasible composition is realized or the developer decides to abort.

[b]MoSCoE stands for Modeling Web Service Composition and Execution. More information is available at: `http://www.moscoe.org`.
[c]We rely on translators similar to the ones proposed in Refs. 39 and 50 to translate existing Web service specifications (e.g., BPEL,[2] OWL-S[21]) to their corresponding STS representations.
[d]Note that determination of the cause for failure of composition, and use of that information for reformulation of the goal specification is carried out at *design-time* as opposed to *run-time*.

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 integrated according to a certain order and their data (and process) flow is modeled in an infinite domain.
- A technique for determining the cause for failure of composition to assist the service developers 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 of our work. Section 3 presents a logical formalism for determining feasible composition strategies and provides theoretical results of our approach. Section 4 introduces how cause(s) for failure of composition are identified such that appropriate reformulation of the goal specification can be carried out by the service developer. Section 5 discusses the implementation of our composition framework in a tabled-logic programming environment.[44] 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`,[e] which allows potential clients to order and buy books 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 considerations: (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 reason for building `KogoBuy` is to allow the client to interact with `KogoBuy` directly as opposed to interacting with the individual components for purchasing a book and shipping it to a particular destination.

To model such a scenario in MoSCoE, the user (service developer) is required to provide a goal service specification using a state machine (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

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

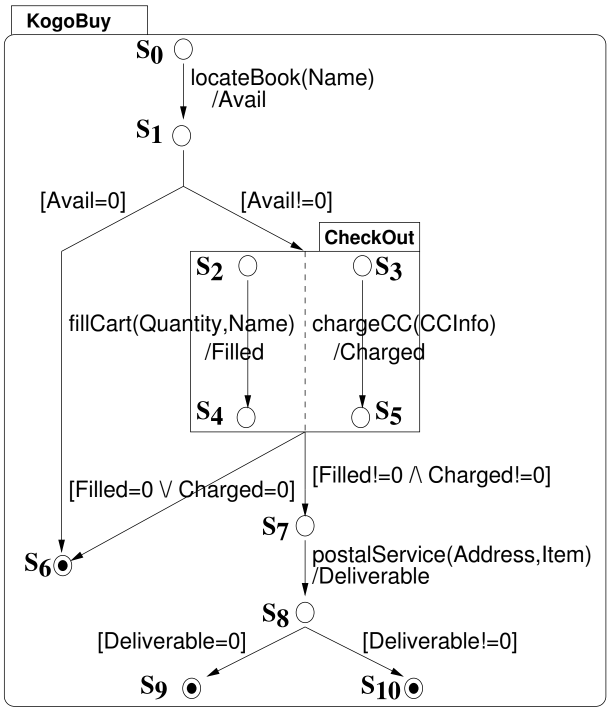


Fig. 1. State Machine representation of `KogoBuy`.

to determine a feasible composition strategy that provides the desired functionality. Figure 2 shows the transition system of `KogoBuy`, and Figures 3(a) and 3(b) show the transition systems of `BookPurchase` and `e-Postal` services, respectively.

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 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'` first authenticates the card and then charges a fee for providing the authentication service. Thus, `CCV'` has an additional behavior in terms of charging for the service that it provides. For a client whose goal is only to validate the authenticity of a card, the additional action is unwarranted. Consequently, we ignore services such as `CCV'` for determining a

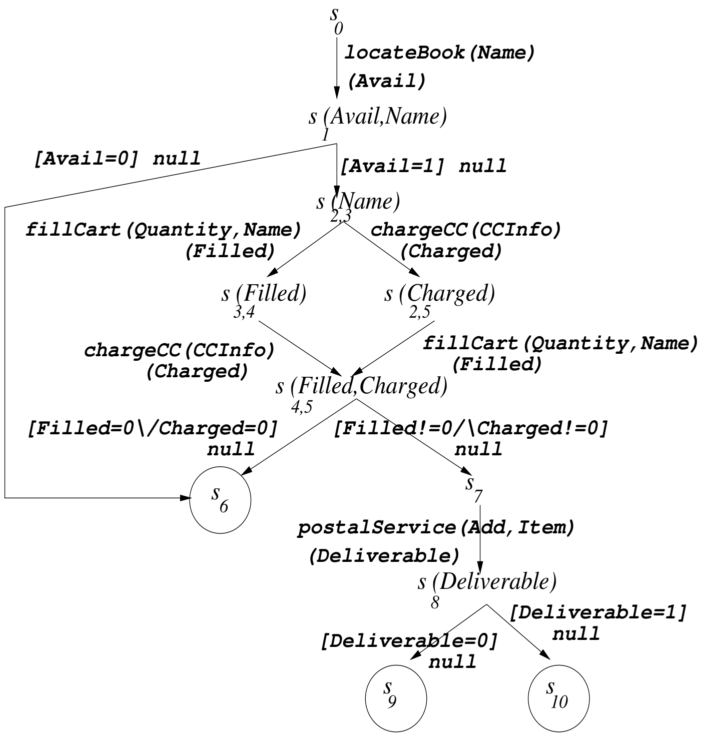


Fig. 2. Symbolic Transition System (STS) representation of KogoBuy.

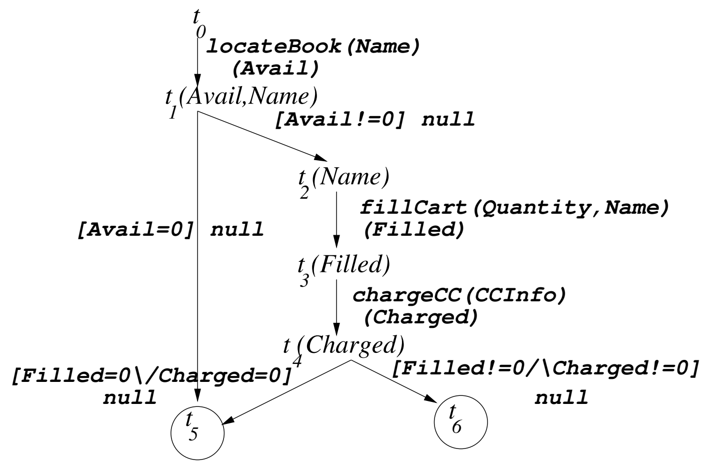
composition strategy from the available pool of services because if we replace CCV with CCV', an additional behavioral functionality is added which is not required (and hence, modeled) by the goal. Note that this approach is in stark contrast to the traditional mediator-based techniques where actions 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 to as *goal-directed service composition.* MoSCoE solves the composition problem using the notion of *realizability* (see Definition 3.3) — a composition of components is said to provide the functionality desired by the goal service if the former *realizes* the goal, 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) that consists of states $(s_1, s_2, \ldots)$ representing an abstraction of the system configuration, and inter-state transitions $(s_1 \longrightarrow s_2)$ denoting the conditions under which

(a)



(b)

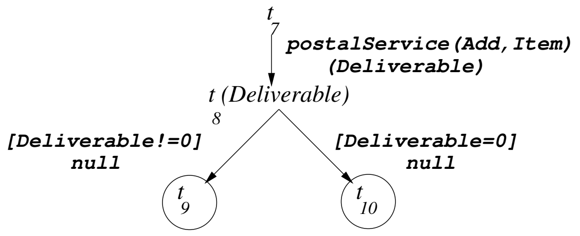


Fig. 3. STS representation of (a) `BookPurchase`, (b) `e-Postal`.

the system evolves from one state to the next. The states can be either *composite* (or-/and- states)[f] or *atomic*.

Each transition, `source` $\xrightarrow{ev[g]/e}$ `destination`, is annotated with *action* labels consisting of an *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; 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, 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`. Note that in our approach, we assume that the control flow of the service

[f]A composite state is an "or-state" if there exists multiple transitions originating from the state such that any one of these transitions can be executed; whereas the composite is an "and-state" if all the transitions can be executed simultaneously.

represented in a state machine is dependent entirely on the guards augmenting the corresponding transitions. That is, if there exists multiple transitions originating from a particular state, then the branching behavior is based on how the guards are analyzed. For example, in Figure 1, there are two transitions originating from state s_1 namely, $s_1 \xrightarrow{[Avail=0]} s_6$ and $s_1 \xrightarrow{[Avail!=0]} s_2s_3$. Thus, the transition from state s_1 to s_6 is executed only when $Avail = 0$, whereas the transition from s_1 to s_2s_3 is executed when $Avail! = 0$. Furthermore, we assume that if there is no correlation between the order in which the events (or functions) are executed (i.e., the functions can be invoked in parallel since there is no dependency between them), they are represented using "and-composite" states (see below for an example) in the state machine model.

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.[11] For example, in Figure 1 the state s_2s_3 represents a composite and-state `CheckOut`. During execution of `KogoBuy`, once the state `CheckOut` is reached (i.e., `CheckOut` is active), the transitions $s_2 \longrightarrow s_4$ and $s_3 \longrightarrow s_5$ can be carried out in parallel. Consequently, the exact sequence or the order in which transitions originating from composite state s_2s_3 are executed cannot be determined apriori. To address the need to model Web services in such a setting, we use *Symbolic Transition Systems* (STS)[4] 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 are represented by a set $vars(t)$. Substitutions, denoted by σ, map 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 3.1. (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, $s0 \in S$

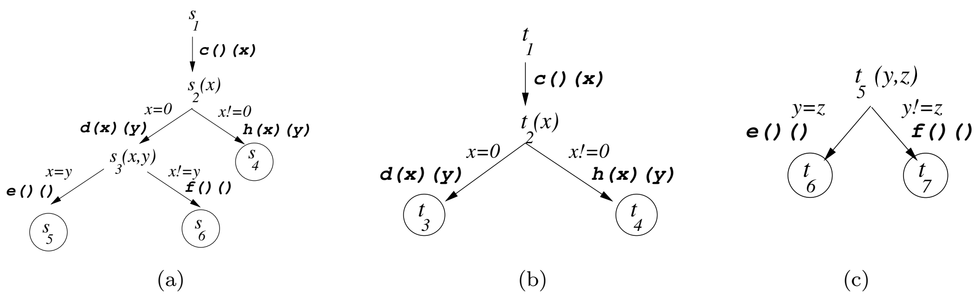


Fig. 4. Example Symbolic Transition Systems. (a) $\mathtt{STS}_g$ (b) $\mathtt{STS}_1$ (c) $\mathtt{STS}_2$.

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 atomic-functions provided by the service and are of the form $a(\vec{x})(y)$ where $\vec{x}$ represents the input parameters and y denotes the return valuations. Whenever, the atomic-function is `null`, i.e., the transition from state s to t is not augmented with α, we represent it with ϵ.
(3) ρ relates $vars(s)$ to $vars(t)$.

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 atomic-function being invoked, `args` is the list of actual input 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 $\gamma\sigma = \mathtt{true}$ and $\gamma \Rightarrow \rho$. The transition under substitution σ is denoted by $s\sigma \xrightarrow{\alpha\sigma} t\sigma$.

3.3. *Composition of symbolic transition systems*

Our objective is to select a set of component services (represented as Symbolic Transition Systems defined above) and compose them in a manner that will realize the desired goal service functionality. We proceed to define composition of STSs.

Definition 3.2. (Composition) Given two STSs, $\texttt{STS}_i = (S_i, \longrightarrow_i, s0_i, S_i^F)$ and $\texttt{STS}_j = (S_j, \longrightarrow_j, s0_j, S_j^F)$, their composition denoted by $\texttt{STS}_i \circ \texttt{STS}_j = (S_k, \longrightarrow_k, s0_k, S_k^F)$ is defined as follows:

(i) $S_k = S_i \cup S_j$

(ii) $s0_k = s0_i$

(iii) $S_k^F = S_j^F$

(iv) $$s_k \xrightarrow{\gamma,\alpha,\rho}_k s_{k'} \Leftarrow \begin{cases} s_i \xrightarrow{\gamma,\alpha,\rho}_i s_{i'} \wedge s_k = s_i \wedge s'_k = s'_i \\ s_j \xrightarrow{\gamma,\alpha,\rho}_j s_{j'} \wedge s_k = s_j \wedge s'_k = s'_j \\ s_k \in S_i^F \ \wedge \ s0_j \xrightarrow{\gamma,\alpha,\rho}_j s_{j'} \wedge s_k = s0_j \wedge s_{k'} = s_{j'} \end{cases}$$

In other words, $\texttt{STS}_i \circ \texttt{STS}_j$ is obtained by merging the final states of $\texttt{STS}_i$ with the start state of $\texttt{STS}_j$ such that every out-going transition of the start state of $\texttt{STS}_j$ is also the out-going transition of each final state of $\texttt{STS}_i$.

Recall that our approach 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 functionality desired by the former can be provided by composing the components $\texttt{STS}_i, \texttt{STS}_j, \ldots \texttt{STS}_k$ such that $\texttt{STS}_g$ is *realized* by $\texttt{STS}_i \circ \texttt{STS}_j \circ \ldots \texttt{STS}_k$. In essence, *realizability* ensures that the composition can 'mimic' the goal service functionality by appropriately modeling its behavior and data/process flow. We proceed to define realizability in the context of STSs required to identify a feasible composition as described above.

Definition 3.3. (Realizability Relation) Given an STS $\mathcal{S} = (S, \longrightarrow, s0, S^F)$, the realizability relation with respect to substitution θ, denoted by $\mathcal{R}^{\,\theta}$, is a subset of $S \times S$ such that

$$s_1 \mathcal{R}^{\,\theta} s_2 \Rightarrow (\exists s_2 \xrightarrow{\gamma,\epsilon,\rho} t_2 \Rightarrow \forall s_2\theta \xrightarrow{\epsilon} t_2\theta . \exists s_1\theta \xrightarrow{\epsilon} t_1\theta \ \wedge t_1 \mathcal{R}^{\,\theta\sigma} t_2)$$
$$\bigwedge \ (\not\exists s_2 \xrightarrow{\gamma,\epsilon,\rho} t_2 \Rightarrow \forall s_1\theta \xrightarrow{\alpha_1} t_1\theta . \exists s_2\theta \xrightarrow{\alpha_2} t_2\theta .$$
$$\forall \sigma . \alpha_1\theta\sigma = \alpha_2\theta\sigma \wedge \ t_1 \mathcal{R}^{\,\theta\sigma} t_2) \, .$$

Two states, under the substitution θ over state variables, are realizable if they are related by the *largest* realizability relation $\mathcal{R}^{\,\theta}$. We say that an $\texttt{STS}_i$ realizes $\texttt{STS}_j$, denoted by $(\texttt{STS}_i \mathcal{R}^{\,\theta} \texttt{STS}_j)$ `iff` $(s0_i \mathcal{R}^{\,\theta} s0_j)$.

Note that in the above relation, whenever there exists a transition $s_2 \longrightarrow t_2$ such that the corresponding transition-action is `null` (denoted by ϵ), then for each such transition originating from state s_2, there must be a corresponding transition

$s_1 \longrightarrow t_1$ originating from state s_1 (with `null` action) such that t_1 realizes t_2. On the other hand, if $s_2 \longrightarrow t_2$ does *not* have a `null` action, then for every transition $s_1 \longrightarrow t_1$ originating from state s_1, there must be a corresponding transition $s_2 \longrightarrow t_2$ such that the transitions are augmented with equivalent actions and t_1 realizes t_2. This notion of realizability ascertains that if $\mathtt{STS}_i$ realizes $\mathtt{STS}_j$, then $\mathtt{STS}_i$ has the appropriate branching as well as functional behavior that is modeled in $\mathtt{STS}_j$.

For example, consider the STSs in Figure 4(a) and 4(b). If state t_1 realizes s_1, then $t_2(x)$ realizes $s_2(x)$ 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, $t_2(x)$ realizes $s_2(x)$ for valuations of x in both these partitions. When $x = 0$, t_3 realizes $s_3(x, y)$ for all possible valuations of y, whereas for $x \neq 0$, t_4 realizes s_4 for all possible valuations of y. The above can be represented using logical expressions as follows:

$$\begin{aligned} t_1 \mathcal{R}^{\mathtt{true}}\ s_1 &\Rightarrow \forall x.(t_2(x)\ \mathcal{R}^{x}\ s_2(x)) \\ &\Rightarrow (t_2(x)\ \mathcal{R}^{x=0}\ s_2(x))\ \wedge\ (t_2(x)\ \mathcal{R}^{x\neq 0}\ s_2(x)) \\ &\Rightarrow (\ \forall y.((t_3\ \mathcal{R}^{x=0,y} s_3(x,y))\ \wedge\ (t_4\ \mathcal{R}^{x\neq 0,y} s_4))\ )\,. \end{aligned} \tag{1}$$

Here, WLOG we assumed that the variable names (x and y) in the two STSs are identical for simplicity. Note that the realizability of $\mathtt{STS}_g$ by $\mathtt{STS}_1$ leads the former to states $s_3(x, y)$ and s_4 with the constraints $x = 0$ and $x \neq 0$, respectively. Thus, a selected component (realizing the goal) drives the goal to some specific states. To identify the states in the goal that are realized by the (final) states of the component, we later define the termination relation (Definition 2).

Definition 3.4. (Realization) Given a goal service $\mathtt{STS}_g$, we say that $\mathtt{STS}_g = (S_g, \longrightarrow_g, s0_g, S_g^F)$ is realized by the composition of available services $\mathtt{STS}_1 \circ \ldots \circ \mathtt{STS}_n$ where $\mathtt{STS}_i = (S_i, \longrightarrow_i, s0_i, S_i^F)$ if and only if $\mathtt{STS}_1 \circ \ldots \circ \mathtt{STS}_n\ \mathcal{R}^{\mathtt{true}}\ \mathtt{STS}_g$ and final states of $\mathtt{STS}_n$ are realizability equivalent to final states of $\mathtt{STS}_g$.

Essentially, the above definition states that if a sequence of n component services composed together realizes the goal service $\mathtt{STS}_g$, then the final state(s) of the nth-component service will be related by the relation $\mathcal{R}^{\theta}$ to the final state(s) in $\mathtt{STS}_g$, where θ is the set of variable substitutions. Thus, during execution when the final state(s) of the nth-component service is reached, thereby terminating its execution, it also implies that the execution of $\mathtt{STS}_g$ has terminated.

We now proceed with the definition of termination relation which identifies the final states in a component service that realizes the states in the goal service.

3.4. *Identifying a realizable composition*

Definition 3.5. (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

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

In the above, t represents the states reached after the realization, 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 realizes s_2 if the final state reached from s_1 realizes state t reachable from s_2 under the constraint δ. We say that ($\mathtt{STS}_i\ \mathcal{T}_t^{\theta,\delta}\ \mathtt{STS}_j$) $\mathtt{iff}$ ($s0_i\ \mathcal{T}_t^{\theta,\delta}\ s0_j$).

Returning to the example in Figure 4(a) and 4(b), state t_1 realizes s_1 and this drives t_1 to $t_2(x)$ and s_1 to $s_2(x)$, where $t_2(x)$ realizes $s_2(x)$ for all possible valuations of x ($x \neq 0$ and $x = 0$). For $x = 0$, there is a transition from state $t_2(x)$ to t_3 and this transition realizes the transition from $s_2(x)$ to $s_3(x, y)$. On the other hand, for $x \neq 0$, there is a transition from $t_2(x)$ to t_4 which realizes the transition from $s_2(x)$ to s_4. Since both t_3 and t_4 are final states of $\mathtt{STS}_1$, the states $s_3(x, y)$ and s_4 are identified as the states realized by the state t_3 under the constraint $x = 0$ and t_4 under the constraint $x \neq 0$, respectively. Thus,

$$\begin{aligned} t_1\ \mathcal{T}_t^{\mathtt{true},\delta}\ s_1 &\Leftarrow t_1\ \mathcal{R}^{\mathtt{true}}\ s_1\ \wedge\ \exists x.(t_2(x)\ \mathcal{T}_t^{x,\delta}\ s_2(x)) \\ &\Leftarrow \mathtt{true}\ \wedge\ (\ (t_2(x)\ \mathcal{T}_t^{x=0,\delta}\ s_2(x))\ \vee (t_2(x)\ \mathcal{T}_t^{x\neq 0,\delta}\ s_2(x))\ ) \\ &\Leftarrow \mathtt{true}\ \wedge (\ \exists y.((t_3\ \mathcal{T}_t^{\{x=0,y\},\delta} s_3(x,y))\ \wedge\ (t_4\ \mathcal{T}_t^{\{x\neq 0,y\},\delta} s_4))\ ) \\ &\Leftarrow \mathtt{true}\ \wedge\ t = \{s_3(x,y), s_4\}\ \wedge\ \delta = \{x = 0, x \neq 0, y\} \end{aligned} \quad (2)$$

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

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

That is, the composition is modeled 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 are realized by the final state of the component under consideration are final states.

$$\begin{aligned} T_n &= \{t_n\theta_n \mid \forall t_{n-1}\theta_{n-1} \in T_{n-1}.\exists t_n\theta_n.(\mathtt{STS}_n\ \mathcal{T}_{t_n}^{\theta_{n-1},\theta_n}\ t_{n-1})\} \\ &\wedge \{t_n \mid t_n\theta_n \in T_n\} \equiv S_g^F . \end{aligned} \quad (4)$$

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

For example, in Figure 4, $t_1\ \mathcal{T}^{\texttt{true},\{x=0,x\neq 0,y\}}_{s_3(x,y),s_4}\ s_1$ (from Eq. (2)). Proceeding further, we select $\texttt{STS}_2$ (Figure 4(c)). The start state of $\texttt{STS}_2$ realizes the state $s_3(x,y)$ under the constraint $x=0$ and the states in $\texttt{STS}_g$ that are realized by the final states of $\texttt{STS}_2$ are s_5 and s_6. Using termination relation, the result is obtained as follows:

$$\begin{aligned}
\forall x &= 0.\forall y.\exists t\delta.\ t_5(x,y)\ \mathcal{T}^{\{x=0,y\},\delta}_{t}\ s_3(x,y)\\
&\Leftarrow t_5(x,y)\ \mathcal{T}^{\{x=0,x=y\},\delta}_{t}\ s_3(x,y)\ \texttt{or}\\
&\quad\ t_5(x,y)\ \mathcal{T}^{\{x=0,x!=y\},\delta}_{t}\ s_3(x,y)\\
&\Leftarrow t_5(x,y)\ \mathcal{T}^{\{x=0,x=y\},\{x=0,x=y\}}_{s_5}\ s_3(x,y)\ \texttt{or}\\
&\quad\ t_5(x,y)\ \mathcal{T}^{\{x=0,x!=y\},\{x=0,x!=y\}}_{s_6}\ s_3(x,y)\,. \qquad (5)
\end{aligned}$$

In the above, states s_4 (obtained from the $\mathcal{T}$ relation in Eq. (2)) and $\{s_5, s_6\}$ (obtained from the $\mathcal{T}$ relation in Eq. (5)) correspond to the final states S^F_g of $\texttt{STS}_g$. Therefore, the composition of $\texttt{STS}_1$ followed by $\texttt{STS}_2$ will provide the desired functionality modeled in the goal $\texttt{STS}_g$.

Theorem 3.1. (Sound and Complete) Given a goal service $\texttt{STS}_g = (S_g, \longrightarrow_g, s0_g, S^F_g)$ and a set of n available component services $\texttt{STS}_1 \ldots \texttt{STS}_n$, where $\texttt{STS}_i = (S_i, \longrightarrow_i, s0_i, S^F_i)$, we say that $\texttt{STS}_1 \circ \ldots \circ \texttt{STS}_n\ \mathcal{R}^{\texttt{true}}\ \texttt{STS}_g$, if and only if $\forall t_n\theta_n \in T_n.t_n \in S^F_g$, where

$$T_i = \begin{cases} \{t_1\theta_1 \mid \exists t_1\theta_1.(\texttt{STS}_1\ \mathcal{T}^{\texttt{true},\theta_1}_{t_1}\texttt{STS}_g)\} & \text{if } i=1\\ \{t_i\theta_i \mid \forall t_{i-1}\theta_{i-1} \in T_{i-1}.\exists t_i\theta_i.(\texttt{STS}_i\ \mathcal{T}^{\theta_{i-1},\theta_i}_{t_i}\ t_{i-1})\} & \text{if } 1 < i \le n \end{cases}$$

Proof Sketch. For $n=1$, the above theorem is trivially true. From Definition 2 it follows that $\texttt{STS}_1\mathcal{T}^{\texttt{true},\theta_1}_{t_1}\texttt{STS}_g$ requires $\texttt{STS}_1\mathcal{R}^{\texttt{true}}\texttt{STS}_g$ such that $t_1 \in S_g$ is realized by $s_1 \in S^F_1$. Thus, when $t_1 \in S^F_g$, the behavior of $\texttt{STS}_g$ is realized by $\texttt{STS}_1$ (Definition 3.4).

In general, $\forall 1 \le i < n.\texttt{STS}_1 \circ \texttt{STS}_2 \circ \ldots \circ \texttt{STS}_i\ \mathcal{T}^{\texttt{true},\theta_i}_{t_i}\ \texttt{STS}_g$ such that $t_i \in S_g$ is realization equivalent to $s_i \in S^F_i$. If the $i+1$-th component is such that $\texttt{STS}_{i+1}\ \mathcal{T}^{\theta_i,\theta_{i+1}}_{t_{i+1}}\ t_i$ for all $t_i\theta_i$ and $t_{i+1} \in S^F_g$, then from Definition 2, $\texttt{STS}_{i+1}$ is realization equivalent to all t_i under the substitution θ_i ($\forall t_i\theta_i.\texttt{STS}_{i+1}\ \mathcal{R}^{\theta_i}t_i$) such that the final states of $\texttt{STS}_{i+1}$ and $\texttt{STS}_g$ are realization equivalent. Thus, the above is true only when $\texttt{STS}_1 \circ \texttt{STS}_2 \circ \ldots \circ \texttt{STS}_{i+1}$ realizes $\texttt{STS}_g$ (Definition 3.4). □

3.5. *Complexity analysis*

The complexity of the composition process is dependent on the complexity of the realizability (Definition 3.3) and termination (Definition 2) relations. Assume that $|S_g|$ is the number of states in the goal $\texttt{STS}_g$, $|S_c|$ is the maximum number of component service states, and n is the total number of available component services. If the composition of the entire set of n components realizes the goal, then complexity of the composition process is $O(|S_g| \times |S_c| \times n)$. However, in the worst case, all the possible 2^n combinations of the component services will have to be analyzed before a failure of composition occurs, thereby resulting in an overall complexity of $O(|S_g| \times |S_c| \times n \times 2^n)$ for our composition algorithm.

3.6. *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 transition system of `KogoBuy` corresponding to its state machine representation (Figure 1). Figures 3(a) & 3(b) show the transition system representation for the component services `BookPurchase` and `e-Postal`, respectively. To determine whether the functionality desired by `KogoBuy` can be achieved from `BookPurchase` and `e-Postal` services, we need to find out if $\texttt{STS}_{\texttt{KB}}$ is realized by the composition of $\texttt{STS}_{\texttt{BP}}$ and $\texttt{STS}_{\texttt{eP}}$. If the component `BookPurchase` is selected first, it can be seen that $t_0\,\mathcal{R}^{\texttt{true}}s_0$ holds, where $t_0 \in \texttt{STS}_{\texttt{BP}}$ and $s_0 \in \texttt{STS}_{\texttt{KB}}$. This is because the transition path starting from state s_0 in `KogoBuy` is realized by the path in `BookPurchase` such that s_6 is the terminal state corresponding to state t_5, and similarly s_7 is the terminal state corresponding to state t_6. In other words, we have $t_0\ \mathcal{T}_{s_6}^{\texttt{true},(Filled=0\|Charged=0)}\ s_0$ and $t_0\ \mathcal{T}_{s_7}^{\texttt{true},(Filled!=0\&Charged!=0)}\ s_0$. Proceeding further, we select $\texttt{STS}_{\texttt{eP}}$ which realizes s_7 under the substitution (*Filled!=0 & Charged!=0*). Thus, the composition of $\texttt{STS}_{\texttt{BP}}$ followed by $\texttt{STS}_{\texttt{eP}}$ will provide the desired functionality as modeled in the goal service $\texttt{STS}_{\texttt{KB}}$.

However, in practice, there might be many component services available in a service repository that might provide the functionality (perhaps in-part) required by $\texttt{STS}_{\texttt{KB}}$. Consequently, it is important to determine all possible combinations of composition of those services that could potentially realize the goal functionality, and allow the service developer to use/deploy one of the candidate compositions (e.g., based on whether non-functional requirements such as Quality of Service are met or not[31]). In light of this, let us assume that, in addition to the `BookPurchase` and `e-Postal` services shown in Figure 3, we also have access to few more component services, namely `LocateBook` ($\texttt{STS}_{\texttt{LB}}$), `LocateBook'` ($\texttt{STS}_{\texttt{LB'}}$) and `ItemPurchase` ($\texttt{STS}_{\texttt{IP}}$), as shown in Figure 5, where the `LocateBook` and `LocateBook'` services allow clients to search for books given their name(s) as input and the `ItemPurchase` service enables clients to fill a virtual cart with items of interest that could be purchased using a credit card. Further assume that, the

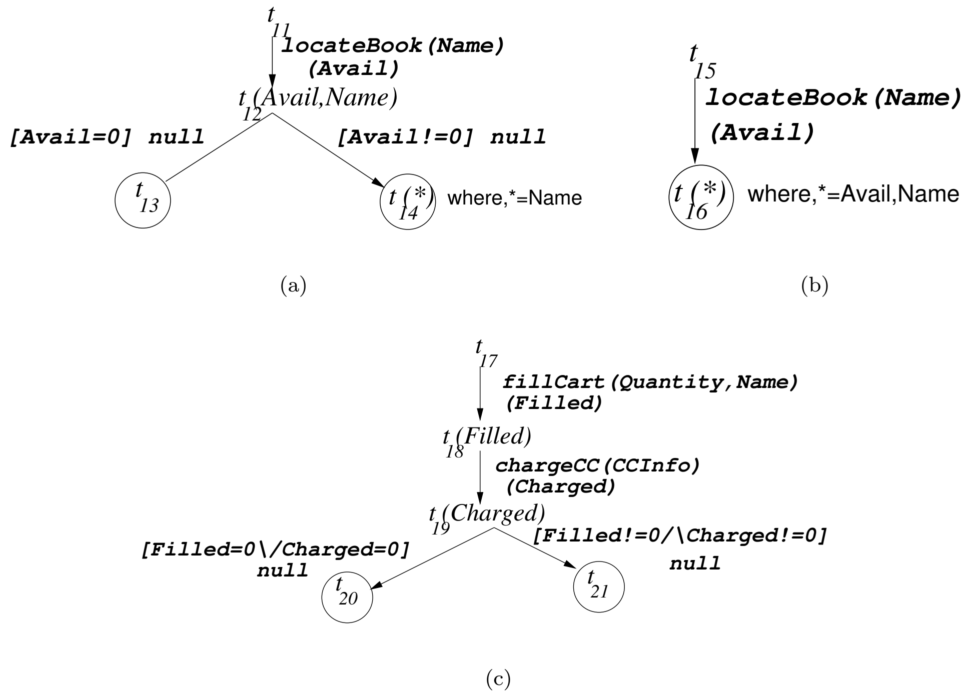


Fig. 5. STS representation of (a) `LocateBook`, (b) `LocateBook'` and (c) `ItemPurchase`.

`LocateBook` (Figure 5(a)) service is selected first by our system[g] and we want to identify if it can be used to realize the goal service $\texttt{STS}_{\texttt{KB}}$? As it can be seen, the relation $t_{11}\,\mathcal{R}^{\texttt{true}}s_0$ holds since the transition paths starting from state s_0 in `KogoBuy` are realized by the paths in `LocateBook` such that s_6 is the terminal state corresponding to state t_{13}, and $s_{2,3}$ is the terminal state corresponding to the state t_{14} in $\texttt{STS}_{\texttt{LB}}$. Furthermore, in the next step of the composition process, if `ItemPurchase` service is selected, it realizes state $s_{2,3}$ in $\texttt{STS}_{\texttt{KB}}$ under the substitution $(Avail{=}1)$ wherein s_6 and $s_{4,5}$ will be the terminal states corresponding to states t_{20} and t_{21} in $\texttt{STS}_{\texttt{IP}}$, respectively. Proceeding even further, if `e-Postal` service is selected next, it realizes $\texttt{STS}_{\texttt{KB}}$ (in-part) as shown earlier. Thus, the composition of $\texttt{STS}_{\texttt{LB}}$, $\texttt{STS}_{\texttt{IP}}$ and $\texttt{STS}_{\texttt{eP}}$ will provide the desired functionality of $\texttt{STS}_{\texttt{KB}}$. However, during this composition process, it is possible that the `LocateBook'` service is selected in lieu of `LocateBook`, since it provides the same functionality as the latter. But, selection of $\texttt{STS}_{\texttt{LB'}}$ does not lead to a feasible composition because even though $t_{15}\,\mathcal{R}^{\texttt{true}}s_0$

[g]At present, the system does an exhaustive search for a set of available component services that can provide the atomic-function modeled in the goal service specification. If there are multiple services that offer the same functionality, any one of those services is selected randomly. In future, we plan to incorporate selection of services by considering non-functional properties such as Quality of Service (QoS).[31]

holds, there exists no component service with transitions that can realize transitions originating from state s_1 in $\texttt{STS}_{\texttt{KB}}$, i.e., $\nexists t_x . t_x \, \mathcal{R}^{\texttt{true}} s_1$ holds, where t_x is the start state in one of the component services. We refer to such scenarios as failures of composition, the identification of which is vital when modeling complex composite services. We discuss the details in the next section.

4. Composition 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 (i.e., the service developer) 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 realize one or more states of the goal $\texttt{STS}_g$ by a component is obtained by identifying the *dual* of greatest fixed point realizability relation $\mathcal{R}$:

$$\begin{aligned} s_1 \, \overline{R}^{\theta} \, s_2 \Leftarrow & (\exists s_2 \xrightarrow{\gamma,\epsilon,\rho} t_2 \ \wedge \ (\exists s_2\theta \xrightarrow{\epsilon} t_2\theta . \forall s_1\theta \xrightarrow{\epsilon} t_1\theta \ \Rightarrow \ t_1 \, \overline{R}^{\theta\sigma} \, t_2) \ ) \\ & \bigvee \ (\nexists s_2 \xrightarrow{\gamma,\epsilon,\rho} t_2 \ \wedge \ (\exists s_1\theta \xrightarrow{\alpha_1} t_1\theta . \forall s_2\theta \xrightarrow{\alpha_2} t_2\theta . \\ & \qquad \exists \sigma . (\alpha_1\theta\sigma = \alpha_2\theta\sigma) \Rightarrow \ t_1 \, \overline{R}^{\theta\sigma} t_2)) \, . \end{aligned} \tag{6}$$

Two states are said to be *not* realizable 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 Eq. (6), the cause of the state s_1 *not* realizing state s_2 can be due to:

(1) $\exists \sigma . \alpha_1\theta\sigma \neq \alpha_2\theta\sigma$ (i.e., the 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, or
(4) $\exists s_2\theta \xrightarrow{\epsilon} t_2\theta$, but there is no transition enabled from s_1 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* realizable:

$$\begin{aligned} s_1 \, \overline{R}_f^{\theta} \, s_2 \Leftarrow & \big(\exists s_2 \xrightarrow{\gamma,\epsilon,\rho} t_2 \wedge (\exists s_2\theta \xrightarrow{\epsilon} t_2\theta . \forall s_1\theta \xrightarrow{\epsilon} t_1\theta . \exists \sigma . (t_1 \overline{R}_f^{\theta\sigma} t_2) \\ & \qquad\qquad \vee \ (\nexists s_1\theta \xrightarrow{\epsilon} \ \wedge \ f = (s_1, s_2, \theta)) \ )\big) \\ & \bigvee \\ & \big(\ \nexists s_2 \xrightarrow{\gamma,\epsilon,\rho} t_2 \wedge (\exists s_1\theta \xrightarrow{\alpha_1} t_1\theta . \forall s_2\theta \xrightarrow{\alpha_2} t_2\theta . \\ & \qquad \exists \sigma . (\ (\alpha_1\theta\sigma \neq \alpha_2\theta\sigma \ \wedge f = (s_1, s_2, \sigma)) \\ & \qquad \vee \ (\alpha_1\theta\sigma = \alpha_2\theta\sigma \ \wedge t_1 \overline{R}_f^{\theta\sigma} t_2) \ ) \\ & \qquad \vee \ (\nexists s_2\theta \longrightarrow \ \wedge \ f = (s_1, s_2, \theta)) \ )\big) \, . \end{aligned} \tag{7}$$

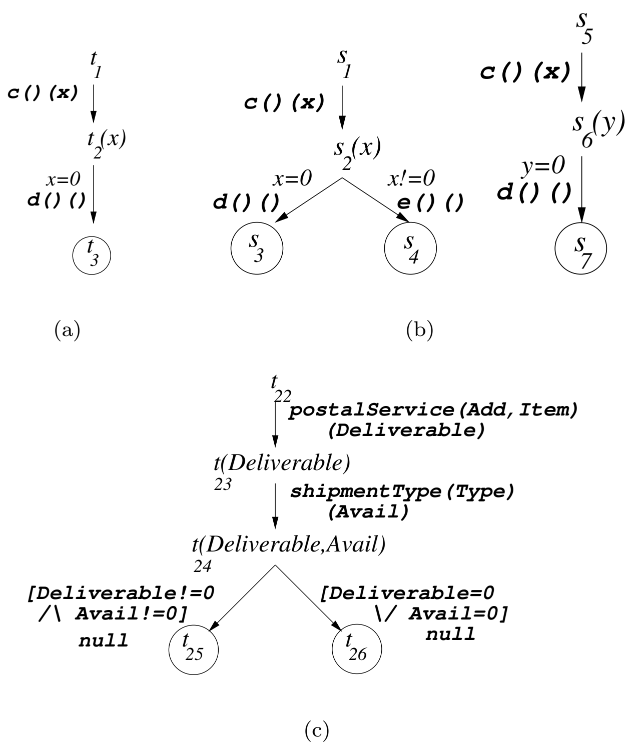


Fig. 6. (a) Component STS, (b) Goal STSs $\mathtt{STS}_g$ & $\mathtt{STS}_{g'}$, (c) STS for NewPostal.

For example, consider the STSs in Figures 6(a) & 6(b). The first $\mathtt{STS}_g$ (rooted at s_1) is not realized by the component STS as there exists a transition from $s_2(x)$ to s_4 when the valuation of x is not equal to zero, which is absent from the corresponding state $t_2(x)$ in the component. That is,

$$\begin{aligned} t_1 \overline{\mathcal{R}}_f^{\mathtt{true}} s_1 &\Leftarrow \exists x. t_2(x)\ \overline{\mathcal{R}}_f^{x}\ s_2(x) \\ &\Leftarrow (t_2(x)\ \overline{\mathcal{R}}_f^{x!=0}\ s_2(x))\ \mathtt{or}\ (t_2(x)\ \overline{\mathcal{R}}_f^{x=0}\ s_2(x)) \\ &\Leftarrow \mathtt{false}\ \mathtt{or}\ (t_2(x)\ \overline{\mathcal{R}}^{x=0}_{(t_2(x),s_2(x),x!=0)}\ s_2(x))\,. \end{aligned}$$

The state t_1 also does not realize state s_5 of $\mathtt{STS}_{g'}$ as the state $t_2(x)$ does not realize $s_6(y)$. 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 realized 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, $t_2(x)$ does not realize $s_6(y)$.

4.1. *Failure-cause analysis for `KogoBuy` composite service*

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 6(c)), which functions exactly like `e-Postal`, but additionally asks the client to provide information about the shipment type (e.g., overnight, 2nd day air, ground). However, since this 'additional' shipment type transition is not present in $\texttt{STS}_{\texttt{KB}}$, the component start state t_{22} does not realize the goal service state s_7,. This information about the transition and substitution causing the failure of realization can be obtained using the $\overline{\mathcal{R}}_f^{\theta}$ relation (see Eq. (7)) as follows:

$$\begin{aligned} t_{22}\ \overline{\mathcal{R}}_f^{\texttt{true}}\ s_7 &\Leftarrow \exists x.t_{23}(x)\ \overline{\mathcal{R}}_f^{x}\ s_8(x), \textit{where } x = \textit{Deliverable} \\ &\Leftarrow t_{23}(x)\ \overline{\mathcal{R}}^{x}_{(t_{23}(x),s_8(x))}\ s_8(x)\,. \end{aligned}$$

As noted in Section 3.6, assuming that the `LocateBook'` service (Figure 5(b)) is selected first, then for all the available component services C (as shown in Figures 3 & 5), $\forall t_x.t_x\ \overline{\mathcal{R}}_f^{\texttt{true}}\ s_1$, where $s_1 \in \texttt{STS}_{\texttt{KB}}$, t_x is the start state for one of the component services in C, and $f = \{s_1(\text{Avail,Name}), t_x\}$. This is because none of the component services has transitions originating from their start state that will realize the transitions originating from state s_1 in $\texttt{STS}_{\texttt{KB}}$ (see Figure 2). As such, selecting `LocateBook'` service will always result in failure of composition.

Note that this kind of failure-cause information can be provided to the user (i.e., the service developer) which can be used for reformulating the goal specification in an iterative manner. For example, in the case with `NewPostal` service, the developer 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 developer decides to abort.

5. Prototype Implementation

We have implemented a prototype of MoSCoE in the XSB[44] tabled-logic programming environment. The core of the implementation consists of encoding the realizability and termination relations, and developing a *meta-interpreter*[h] 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 realizability checker, where states and transitions are explored only if they are needed to prove or disprove realizability. In what follows, we proceed with a brief introduction to XSB (Section 5.1) followed by the 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.moscoe.org`.

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

5.1. *Preliminaries: XSB tabled-logic programming*

XSB logic programming system,[44] developed at SUNY Stony Brook, is an extension of Prolog-style SLD resolution with tabling.[i] 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., $\texttt{G1} \wedge \texttt{G2} \wedge \cdots \wedge\ \texttt{Gn} \Rightarrow \texttt{G}$. 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, s0).
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 `s0`, (b) `s0` to `s1`, (c) `s1` to `s0` and (d) `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.

[i]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.

5.2. *Logical encoding of service composition*

5.2.1. *Transition relations*

As outlined, an important element of our implementation is encoding the realizability and termination relations, 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.

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 realizability*

The relation described in Eq. (6) can be encoded as a logic program and interpretation of the least model of the program provides the solution to the corresponding relation.

```
:- table negateR/2.

negateR(S1, S2) :-
   trans(S2, _, epsilon, _),
   late_trans(S2, A2, T2),
   no_matching_trans_1(S1, A2, T2).
```

```
negateR(S1, S2) :-
   not trans(S2, _, epsilon, _),
   late_trans(S1, A1, T1),
   no_matching_trans_2(S2, A1, T1).

no_matching_trans_1(S1, A2, T2) :-
   forall( (A1,T1),
           late_trans(S1, A1, T1),
           negateRnext(A1, T1, A2, T2) ).


no_matching_trans_2(S2, A1, T1) :-
   forall( (A2,T2),
           late_trans(S2, A2, T2),
           negateRnext(A1, T1, A2, T2) ).

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

In the above, meta-interpretation of `negateR`, i.e., `interp(negateR(S1, S2), Cin, Cout)` represents the $\overline{\mathcal{R}}^{\theta}$ relation where `Cin` represents θ. The above program provides two rules for `negateR` — the first corresponds to first disjunct of Eq. (6) and the second corresponds to the second disjunct. The first rule states that, if there is an epsilon transition from `S2` which is not matched by any transition from `S1` (predicate `no_matching_trans_1`), then `S1` is *not* realized by `S2`. The predicate `no_matching_trans_1` aggregates all the transitions from `S1` and invokes `negateRnext` on each element of the aggregation (using predicate `forall`). Hence, in `negateRnext`, 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 Eq. (6) (note $\exists\sigma.(\alpha_1\theta_1\sigma = \alpha_2\theta_2\sigma) \Rightarrow t_1\ \overline{R}^{\theta\sigma}\ t_2$). The predicate `negateRnext`, therefore, evaluates to true (a) when `A11=A22` and the subsequent states are not similar; or (b) `A11\=A22`. The second rule for `negateR` can be explained in similar fashion.

The predicate `negateR` can be directly extended to encoding of Eq. (7) required to identify the cause of failure of realizability.

5.2.3. *Termination relation*

The Definition 2 is encoded using the above encoding of `negateR`. 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 negateR(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 realizable (negation of negateR), if so identify two matching transitions and recursively invoke terminal predicate on the subsequent states. The meta-interpreter interp evaluates the negation of negateR as follows: if interpretation of negateR 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 will realize the goal is obtained using the predicate strategy. In essence, definition of strategy is the encoding of Eq. (3).

```
strategy(GoalStates, StratIn, StratOut) :-
  startcomponent(S1),
  allterminal(S1, GoalState, Terminals),
  ( finalgoal(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 equal to the 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.

We have conducted some preliminary analyses to validate our prototype implementation. These experiments were used 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 and were based on the `KogoBuy` example illustrated in Section 2. Thus, the goal was to model the `KogoBuy` composite service using the set of available services as shown in Figures 3, 5, and 6(c). Recall that there are two possible solutions in this scenario (see Section 3.6): composing `BookPurchase` followed by `e-Postal`, and composing `LocateBook` followed by `ItemPurchase` followed by `e-Postal`. We also ran experiments involving modifications of the available services obtained by deleting transitions. For example, even if the transition from state $t_8(Deliverable)$ to t_{10} in `e-Postal` service (Figure 3(b)) is deleted, its composition with `BookPurchase` service (Figure 3(a)) will realize the `KogoBuy` goal service. In all the above scenarios, a composition was identified in less than 2 seconds.[j] Furthermore, we tried to find a composition using `LocateBook'` and `NewPostal` (see Section 4.1) and their variations, which resulted in the failure of composition. The failure cause was correctly identified within a second. 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. However, these findings are preliminary and a systematic evaluation of the proposed approach on standard benchmarks[26] as well as a real-world application is in progress.

6. Related Work

A variety of approaches to automated service composition based on the planning techniques of artificial intelligence, logic programming, and automata-theory have been developed (see Refs. 12, 18, 25, 27, 41 and 48 for surveys).

Of particular interest in the context of the work described in this paper are approaches to service composition within a transition system based framework. Fu *et al.*[15] model Web services as automata extended with a queue, that communicate by exchanging a sequence of asynchronous messages, to synthesize a composition for a given specification. Their approach is extended in Colombo[7] which models services as labeled transition systems with composition semantics defined via message passing, where the problem of determining a feasible composition (i.e., a mediator) is reduced to satisfiability of a deterministic propositional dynamic logic formula. Our framework has been inspired by, and builds on, insights from Colombo. However, in this work we focused in determining a suitable ordering/sequence in which the component services can be integrated together such that the composition will realize the desired functionality. The issue of automatically generating a mediator has been addressed in our previous work.[33] Pistore *et al.*[37,38,50] represent Web services using non-deterministic state transition systems which communicate

[j]The experiments were carried out in an Ubuntu Linux 6.10 machine with 1GB RAM and Intel Core2 Duo 1.8Ghz processor.

through message passing. This approach constructs a parallel composition of *all* the available component services and then generates a plan that controls the services, based on the functional requirements specified using a temporal logic-based language. In contrast, services (goal and components) in our framework are represented using Symbolic Transition Systems augmented with guards over variables with infinite domains. Although we do not focus in composing services in parallel, we still avoid the expensive step of generating a composition of all the available services (before developing a controller). Essentially, we analyze transitions in the component and goal services as and when needed, thereby constructing the composition on-the-fly.

Many recent efforts have also focused on applying logic-programming techniques to automated service composition. Rao *et al.*[40] translated semantic Web service descriptions in DAML-S[3] to extralogical axioms and proofs in linear logic, and used π-calculus for representing and determining composite services by theorem proving. Waldinger[51] 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[22] adapt and extend Golog logic programming language for automatic construction of composite Web services. The approach allows requesters to specify goals 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[44] 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.

A number of approaches[5,6,17,28] have been proposed which rely on designing models (using standard UML tools) rather than the actual code. Such a Model-Driven Architecture (MDA) for service composition was introduced in Ref. 28. SELF-SERV[6] uses UML state-charts for modeling composite services, which are then declaratively composed and executed in a dynamic peer-to-peer environment. Gronmo *et al.*[17,47] use UML for capturing composite Web service patterns. These patterns, expressed in UML activity diagrams with additional extensions, specify the control flow between the component services realizing the composite service. The model is then translated into an XML-based workflow language for execution. Manolescu *et al.*[20] proposed a declarative model-driven methodology for visualizing, designing and deploying Web applications using Web services. They use a hypertext model (based on WebML[8]) for describing Web interactions and defining specific concepts in the model to represent Web service calls. An MDA-based approach for specifying and executing Semantic Web Services[23] was proposed in Ref. 49. The approach relies on manually specifying semantic Web services using UML specifications which are translated into equivalent OWL-S[21] specifications. However, despite the advances, 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, MoSCoE provides the ability to identify reasons for failure of an attempt to compose a goal service using available components. This, together with its ability to work with abstract (and possibly incomplete) goal service specifications, provide a basis for failure-guided iterative reformulation of the goal service.

In addition to above, many recent approaches have been proposed to facilitate automatic service composition via machine readable descriptions of Web services[23] such as OWL-S[21] and WSMO.[43] By leveraging these developments, Sirin *et al.*[45] proposed a semi-automatic method for Web service discovery and composition. In their system, a user is presented with a set of services which matches the requested service. The choice of the possible services is based on both functional and non-functional aspects. These aspects are presented by OWL classes and an inference engine is used to match the services. Similarly, Paolucci *et al.*[29] proposed locating services based on semantic match between a declarative description of the service sought, and a description of the service offered. These discovered services could then be composed together to realize the goal. On a slightly different note, in Ref. 46 a Hierarchical Task Network (HTN) planner, SHOP2, is used for automatic service composition which are described using OWL-S. Here, the authors provide an algorithm for translating the OWL-S service descriptions into a SHOP2 domain and prove the correctness of their approach by showing the correspondence to the situation calculus semantics of OWL-S. Medjahed and Bouguettaya[24] introduced the *WebDG* system for automatic composition of semantic Web services. *WebDG* adopts a rule-based planning approach where a composite service is generated from a high-level description based on satisfiability of certain constraints specified by the rules (e.g., semantic compatibility). A similar rule-based approach is proposed by Chun *et al.*[9] Here also the authors leverage semantics associated with services (described in DAML-S) to ensure syntactic and semantic compatibility for realizing feasible compositions. While at present, our approach does not take into consideration ontologies and semantics-based service description for determining a composition, we plan to extend our technique along these dimensions by leveraging our previous work on semantic Web service discovery[36] and workflow composition.[35]

7. Summary and Discussion

7.1. *Summary*

We have introduced a novel approach to developing composite services through an iterative reformulation of the goal service specifications. Our technique uses Symbolic Transition Systems (STS) to represent Web services which 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 developers by allowing them to begin with an abstract, possibly incomplete specification of the desired goal that can be modified and reformulated iteratively so as to 'reduce the gap' between the desired functionality and the capabilities of the available components.

7.2. *Discussion and further work*

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 Ref. 19. Additionally, we focused on services which demonstrate a deterministic behavior without loops. Handling nondeterministic behavior that often characterizes real-world services is an important area of ongoing research. We also plan to investigate the complexity of incorporating services with loops in our composition model. We have assumed that the component services are all specified using variable, function, relation names and constants based on common semantics, an assumption that may not hold in the case of services offered by autonomous service providers. In this context, approaches based on inter-ontology mappings to bridge the semantic gap between semantically heterogeneous services[35,36] deserve further investigation. Furthermore, our approach was based on the assumption that the component services are published using Symbolic Transition Systems (STSs). In practice, these specifications can be obtained from service descriptions provided in high-level languages such as BPEL or OWL-S by applying translators proposed in Refs. 38 and 50. However, in our case, these translators will have to be enhanced to handle pre-conditions of atomic functions which correspond to guards in transitions of STSs. Finally, one area that needs significant investigation is the evaluation of our approach using real-world and benchmark[26] cases for service composition. As with any existing technique for service composition, the practical feasibility of our approach is also ultimately limited by the computational complexity of the service composition algorithm. Hence, methods for reducing the number of candidate compositions need to be examined e.g., by exploiting domain specific information to impose a partial-order over the available services, or reducing the number of goal reformulation steps needed by exploiting relationships among failure causes (or between failure causes and services, or between services) need further investigation. In particular, our 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 doing usability studies along those dimensions. Additionally, we plan to investigate the precision and recall measures to capture the effectiveness of search for component services that match the specifications, doing which will require systematic experiments using service composition benchmarks. However, we can draw an analogy with information retrieval systems that respond to user queries (typically expressed using keywords) in a single step as opposed to systems that allow users to iteratively reformulate their query based on the retrieved results.[16,42]

In general, the information retrieval systems that support iterative query reformulation are able to achieve superior performance in terms of precision and recall (relative to documents of interest to the user). Analogously, all other factors being same, the precision and recall achievable (after a few iterations of reformulating the goal) by any service composition system that supports iterative reformulation of specifications would be superior compared to a system that does not.

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`), and NSF grant# 0702758 to Samik Basu, Robyn Lutz and Vasant Honavar.

References

1. G. Alonso, F. Casati, H. Kuna, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, 2004.
2. T. Andrews, F. Curbera, and *et al.* Business Process Execution Language for Web Services, Version 1.1. In *http://www.ibm.com/developerworks/library/ws-bpel/*, 2003.
3. A. Ankolekar, M. Burstein, J. R. Hobbs, and *et al.* DAML-S: Semantic Markup for Web Services. In *International Semantic Web Workshop*, 2001.
4. 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.
5. B. Bauer and M.-P. Huget. Modelling Web Service Composition with UML 2.0. *Intl. Journal of Web Engineering and Technology*, 1(4):484–501, 2004.
6. B. Benatallah, Q. Sheng, and M. Dumas. The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 7(1):40–48, 2003.
7. D. Berardi, D. Calvanese, D. G. Giuseppe, R. Hull, and M. Mecella. Automatic Composition of Transition-based Semantic Web Services with Messaging. In *31st International Conference on Very Large Databases*, pages 613–624, 2005.
8. S. Ceri, P. Fraternali, and A. Bongio. Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. *Computer Networks*, 33(1-6):137–157, 2000.
9. S. A. Chun, V. Atluri, and N. R. Adam. Using Semantics for Policy-Based Web Service Composition. *Distributed and Parallel Databases*, 18(1):37–64, 2005.
10. M. Crane and J. Dingel. On the Semantics of UML State Machines: Categorization and Comparision. In *Technical Report 2005-501, School of Computing, Queen's University, Canada*, 2005.
11. M. L. Crane and J. Dingel. UML Vs. Classical Vs. Rhapsody Statecharts: Not All Models Are Created Equal. In *8th International Conference on Model Driven Engineering Languages and Systems*, pages 97–112. LNCS 3713, 2005.
12. S. Dustdar and W. Schreiner. A Survey on Web Services Composition. *International Journal on Web and Grid Services*, 1(1):1–30, 2005.
13. T. Erl. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall, New Jersey, 2004.
14. D. Ferguson and M. Stockton. Service-Oriented Architecture: Programming Model and Product Architecture. *IBM Systems Journal*, 44(4):753–780, 2005.

15. 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.
16. F. Giannotti and G. Manco. Specifying Mining Algorithms with Iterative User-Defined Aggregates. *IEEE Transactions on Knowledge and Data Engineering*, 16(10):1232–1246, 2004.
17. R. Grønmo and I. Solheim. Towards Modeling Web Service Composition in UML. In *2nd International Workshop on Web Services: Modeling, Architecture and Infrastructure*, pages 72–86, 2004.
18. R. Hull and J. Su. Tools for Composite Web Services: A Short Overview. *SIGMOD Record*, 34(2):86–95, 2005.
19. 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.
20. I. Manolescu, M. Brambilla, S. Ceri, S. Comai, and P. Fraternali. Model-Driven Design and Deployment of Service-Enabled Web Applications. *ACM Transactions on Internet Technology*, 5(3):439–479, 2005.
21. D. Martin, M. Burstein, J. Hobbs, and *et al.* OWL-S: Semantic Markup for Web Services, Version 1.1. In *http://www.daml.org/services/owl-s*, 2004.
22. 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.
23. S. McIlraith, T. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.
24. 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.
25. N. Milanovic and M. Malek. Current Solutions for Web Service Composition. *IEEE Internet Computing*, 8(6):51–59, 2004.
26. S.-C. Oh, H. Kil, D. Lee, and S. R. T. Kumara. WSBen: A Web Services Discovery and Composition Benchmark. In *4th International Conference on Web Services*, pages 239–246. IEEE Press, 2006.
27. S.-C. Oh, D. Lee, and S. Kumara. A Comparative Illustration of AI Planning-based Web Services Composition. *ACM SIGecom Exchanges*, 5(5):1–10, 2005.
28. B. Orriëns, J. Yang, and M. P. Papazoglou. Model Driven Service Composition. In *1st International Conference on Service Oriented Computing*, pages 75–90. LNCS 2910, Springer-Verlag, 2003.
29. M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. In *1st Intl. Semantic Web Conference*, pages 333–347. Springer-Verlag, 2002.
30. J. Pathak. MoSCoE: A Specification-Driven Framework for Modeling Web Services using Abstraction, Composition, and Reformulation. In *2nd IBM Ph.D. Symposium at 4th International Service Oriented Computing Conference*, pages 1–6. IBM Research Technical Report, RC24118, 2006.
31. J. Pathak, S. Basu, and V. Honavar. Modeling Web Services by Iterative Reformulation of Functional and Non-Functional Requirements. In *4th International Conference on Service Oriented Computing*, pages 314–326. LNCS 4294, Springer-Verlag, 2006.
32. 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.

33. J. Pathak, S. Basu, R. Lutz, and V. Honavar. Parallel Web Service Composition in MoSCoE: A Choreography-based Approach. In *4th IEEE European Conference on Web Services*, pages 3–12. IEEE CS Press, 2006.
34. J. Pathak, S. Basu, R. Lutz, and V. Honavar. Selecting and Composing Web Services through Iterative Reformulation of Functional Specifications. In *18th IEEE International Conference on Tools with Artificial Intelligence*, pages 445–454. IEEE CS Press, 2006.
35. J. Pathak, D. Caragea, and V. Honavar. Ontology-Extended Component-Based Workflows-A Framework for Constructing Complex Workflows from Semantically Heterogeneous Software Components. In *2nd International Workshop on Semantic Web and Databases*, pages 41–56. LNCS 3372, Springer-Verlag, 2004.
36. 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.
37. M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated Composition of Web Services by Planning at the Knowledge Level. In *19th Internatinal Joint Conferences on Artificial Intelligence*, pages 1252–1259, 2005.
38. 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.
39. M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated Synthesis of Composite BPEL4WS Web Services. In *3rd IEEE International Conference on Web Services*, pages 293–301. IEEE Press, 2005.
40. J. Rao, P. Kungas, and M. Matskin. Logic-based Web Services Composition: From Service Description to Process Model. In *2nd IEEE International Conference on Web Services*, pages 446–453. IEEE CS Paper, 2004.
41. J. Rao and X. Su. A Survey of Automated Web Service Composition Methods. In *1st Intl. Workshop on Semantic Web Services and Web Process Composition*, pages 43–54, 2004.
42. D. Robins. Interactive Information Retrieval: Context and Basic Notions. *Informing Science*, 3(2):57–62, 2000.
43. D. Roman, U. Keller, H. Lausen, and et al. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.
44. 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.
45. E. Sirin, J. Hendler, and B. Parsia. Semi-automatic Composition of Web Services using Semantic Descriptions. In *Workshop on Web Services: Modeling, Architecture and Infrastructure*, pages 17–24, 2003.
46. E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN Planning for Web Service Composition using SHOP. *Journal of Web Semantics*, 1(4):377–396, 2004.
47. D. Skogan, R. Grønmo, and I. Solheim. Web Service Composition in UML. In *8th IEEE Intl. Enterprise Distributed Object Computing Conference*, pages 47–57. IEEE Press, 2004.
48. M. ter Beek, A. Bucchiarone, and S. Gnesi. Web Service Composition Approaches: From Industrial Standards to Formal Methods. In *2nd International Conference on Internet and Web Applications and Services*, pages 15–20. IEEE CS Press, 2007.
49. J. Timm and G. Gannod. A Model-Driven Approach for Specifying Semantic Web Services. In *3rd International Conference on Web Services*, pages 313–320. IEEE press, 2005.

50. P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *3rd International Semantic Web Conference*, pages 380–394. Springer-Verlag, 2004.
51. R. J. Waldinger. Web Agents Cooperating Deductively. In *1st Intl. Workshop on Formal Approaches to Agent-Based Systems*, pages 250–262. Springer-Verlag, 2001.