Behavior Based Service Composition

Fangzhe Chang[1], Pavithra Prabhakar[2], and Ramesh Viswanathan[1]

[1] Bell Laboratories, Alcatel-Lucent
{fangzhe,rv}@bell-labs.com

[2] University of Illinois at Urbana-Champaign
pprabha2@illinois.edu

Abstract. To enable flexible leveraging of the growing plethora of available web services, service clients should be automatically composed based on required behavior. In this paper, we present a foundational framework for behavior based composition. Services advertise their behavior as labeled transition systems with the action labels corresponding to their externally invocable operations. Query logics are defined in a simple extension of μ-calculus with modalities in which variables are allowed to occur. Query logics specify the desired behavior of the composition with the variables standing for the programs that need to be synthesized. We define a special subclass of programs, called deterministic and crash-free, which behave deterministically (even if the services used are non-deterministic) with all program steps successfully executing in whichever state the services may be in during entire execution. We present an algorithm that synthesises deterministic and crash-free programs whenever there exists such a solution. Since the μ-calculus is the most expressive logic for regular properties, our results yield a complete solution to the automatic composition problem for regular behavioral properties.

1 Introduction

The growing plethora of web services promises to turn the Web from an information repository for human consumption into a world-wide system for distributed computing. A number of communication mechanisms (SOAP [27], REST [11, 12]) for accessing a service's operations over web protocols, and associated formats for exchanging data (WSDL [9], WADL [13]) have been proposed and standardized. Potentially, then, the web can be seen as providing a huge library of components that can be leveraged in building new applications. In the traditional paradigm for programming applications, the developer is required to have a knowledge of and understand the semantics of any components used. Since services are similar to objects in an object-oriented model exporting multiple operations with an expected protocol for the order in which the operations can be invoked, the developer's understanding has to include this service protocol. This form of *manual composition*, in current software development practice, works well when the number of dependent components is small and change infrequently. However, it is less effective in the context of leveraging web services. First, it makes application development more costly and cumbersome by requiring a manual search for any relevant services and an understanding of the behavior of their operations and expected protocol. Second, taking advantage of any new and better services requires an extensive manual effort of performing this search again and recoding the application. Third, the existing implementation of the application may suddenly fail if one of the services used changes even in any small way such as changing the name of one of its operations or its service protocol.

To leverage web services that are numerous and continuously being deployed, merged, and deprecated, one therefore needs a form of *automatic composition*. In this paradigm, client applications are described in terms of their desired behavioral properties that does not require awareness of existing services. The program for the application is then automatically synthesized as a composition of operations from existing services. To support such automatic synthesis, services are required to publish more than their static interfaces — service advertisements specify an abstraction of the behavior of their operations and the protocols to be followed for invoking them.

In this paper, we present a framework and solution towards enabling such automatic behavior based composition. We consider service advertisements expressed as labelled transition systems. The action names in such an advertisement correspond to the exported operations and the propositional constants correspond to externally observable tests on the service state or its outputs. To specify the synthesis requirements of the application, we propose a new logic that is a natural extension of μ-calculus L_μ [7]. Based notably on logics such as the Propositional Dynamic Logic (PDL) [15], it is well understood that modalities can be associated with the programs in terms of the transition relations they induce. The logic we propose includes "unknown" modalities with the unknowns being represented by variables that stand for the programs to be synthesized.

The programs we consider for synthesis can invoke service operations (action names) and test for externally observable outputs (propositional constants) in combination with control constructs such as branching and looping. However, we restrict them to have two properties that we identify as being executionally desirable — we term such programs as being *crash-free* and *deterministic*. The execution steps of deterministic crash-free programs are guaranteed to successfully execute and be deterministic in whichever state the service is driven to. In particular, they are automatically guaranteed to respect the service protocol for invoking operations as manifested in its advertisement.

We present a synthesis algorithm that proceeds in two steps. In the first step, we construct the set of all transition relations that are satisfying solutions for the unknown modalities in a formula. A salient aspect of this step, illustrating an appealing feature of the logic, is that this construction can be defined compositionally in the formula. In the second step, we show how to synthesize a program for any given transition relation. The requirement of being crash-free and deterministic makes this step non-trivial. The resulting synthesis algorithm is complete in that it returns a satisfying solution whenever there exists one. This completeness result holds without any restrictions on the service advertisement — the transition system can be non-deterministic and, furthermore, may not be fully externally observable in that its states do not have to be distinguishable through propositional tests. The synthesis algorithm always yields regular programs — besides establishing decidability, it therefore shows that synthesis requirements expressed in the μ-calculus can be met by finite-state programs even when they are required to be crash-free and deterministic.

2 Formal Framework

Section 2.1 defines the form of behavior advertisements considered in this paper. Section 2.2 defines the class of programs considered for service clients.

2.1 Service Advertisements

The service advertisements we consider are finite labeled transition systems. Let $\mathcal{P}$ be a set of propositional constants. A valuation over $\mathcal{P}$ is a set $\mathbf{v} \subseteq \mathcal{P}$ denoting a truth assignment that assigns true to all propositions $p \in \mathbf{v}$ and false to all $p \notin \mathbf{v}$. We use variables $\mathbf{v}, \mathbf{v}_0, \mathbf{v}_1, \ldots$ to range over propositional valuations.

Definition 1 (Labeled Transition Systems (*LTS*)). *A labeled transition system $\mathcal{T} = (\mathcal{P}, \mathcal{A}, Q, \mathcal{V}, \{\overset{a}{\rightarrow} \mid a \in \mathcal{A}\}, I)$ where $\mathcal{P}$ is a set of propositional constants, $\mathcal{A}$ a set of action names, Q a finite set of states, $\overset{a}{\rightarrow} \subseteq Q \times Q$ is the transition relation for the action name $a \in \mathcal{A}$, $\mathcal{V} : Q \rightarrow 2^{\mathcal{P}}$ is the state valuation function with $\mathcal{V}(q)$ giving the valuation for the propositional constants in any state $q \in Q$, and $I \subseteq Q$ is the set of possible initial states.*

Intuitively, the actions in $\mathcal{A}$ correspond to the operation names that the service makes available. The propositional constants in $\mathcal{P}$ serve as an abstraction for the externally observable properties of operation outputs and the service state. Figure 1 presents an example of an advertisement for a simplified version of an "Amazon"-like shopping service. The operations made available are search s, login l, add-to-cart a, and checkout c. The propositional constants are P_s denoting a successful search result, P_l denoting a successful login, P_a denoting a non-empty cart, and P_c denoting checkout completion. The initial state is q_1. The example illustrates why advertisements are naturally non-deterministic, *e.g.*, a search may or may not result in success. Thus in state q_1, the action s can transition to either state q_3 (when the search is successful) or q_1 (when the search does not yield a result). A similar explanation accounts for the transitions from q_1 on the action l to either q_1 or q_2. After a search yields a result, the item can be added to the cart and checkouts can only be done when the cart is non-empty. While a search can be performed with or without logging in, the advertisement requires that before an item can be added to the cart, one must have logged in. Having logged in at any time, further additions to the cart do not require one to login again.

We will use infix notation for relations writing, e.g., $q \overset{a}{\rightarrow} q'$ for $(q, q') \in \overset{a}{\rightarrow}$. We write $q \overset{a}{\nrightarrow}$ to denote that $\nexists q'. \ q \overset{a}{\rightarrow} q'$ (and $q \overset{a}{\rightarrow}$ otherwise). A (labeled) path

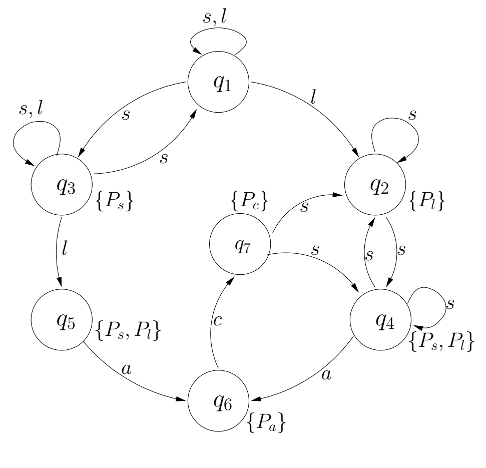

Fig. 1: Service Advertisement Example

π in a transition system is a sequence of the form $q_0 \overset{a_1}{\to} q_1 \cdots \overset{a_i}{\to} q_i \cdots \overset{a_n}{\to} q_n$ with $n \geq 0$ such that $q_i \overset{a_{i+1}}{\to} q_{i+1}$ for $0 \leq i < n$. The trace of the path π is the word $\mathcal{V}(q_0)a_1\mathcal{V}(q_2)a_2 \cdots a_n\mathcal{V}(q_n)$; we use $\mathit{Trace}_{\mathcal{T}}(q_1, q_2)$ to denote the set of words which are traces of paths starting in q_1 and ending in q_2. We will drop the subscript $\mathcal{T}$ when the transition system is clear from the context.

2.2 Programs

The basic operations of client programs are tests of the propositional constants and invocation of action names. Let $\mathcal{P}$ be a set of propositional constants and $\mathcal{A}$ be a set of action names. We define a *program trace* over $(\mathcal{P}, \mathcal{A})$ to be a word in $2^{\mathcal{P}}(\mathcal{A}\,2^{\mathcal{P}})^*$, *i.e.*, a sequence of the form $\mathbf{v}_0a_1\mathbf{v}_1 \ldots a_i\mathbf{v}_i \ldots a_n\mathbf{v}_n$ with $n \geq 0$, $\mathbf{v}_i \subseteq \mathcal{P}$ for $0 \leq i \leq n$, and $a_i \in \mathcal{A}$ for $1 \leq i \leq n$. The operational intuition is as follows: an element a_i corresponds to invoking the action a_i yielding the next state(s), an element $\mathbf{v}_i$ tests whether the propositional valuation in the current state matches $\mathbf{v}_i$, and a program trace is a sequence of such operations. A program is then taken to be a set of such execution traces, *i.e.*, a language $L \subseteq 2^{\mathcal{P}}(\mathcal{A}\,2^{\mathcal{P}})^*$. Following these intuitions, we define programs and the execution of a program with respect to a service as given by its induced transition relation.

Definition 2. *Let $\mathcal{T} = (\mathcal{P}, \mathcal{A}, Q, \mathcal{V}, \{\overset{a}{\to} \mid a \in \mathcal{A}\}, I)$ be a transition system. The set of programs over $\mathcal{P}$ and $\mathcal{A}$ is defined to be the languages of traces $\mathcal{LT}(\mathcal{P}, \mathcal{A}) = \{L \mid L \subseteq 2^{\mathcal{P}}(\mathcal{A}\,2^{\mathcal{P}})^*\}$.*

- *For any finite program trace $w \in 2^{\mathcal{P}}(\mathcal{A}\,2^{\mathcal{P}})^*$, the transition relation induced by w in $\mathcal{T}$, denoted $\mathcal{T}(w)$, is defined by $(q, q') \in \mathcal{T}(w)$ iff there exists a path $q_0 \overset{a_1}{\to} q_1 \cdots \overset{a_i}{\to} q_i \cdots \overset{a_n}{\to} q_n$ such that $q = q_0$, $q' = q_n$ and*

$$w = \mathcal{V}(q_0)\; a_1\; \mathcal{V}(q_1)\; \ldots\; a_i\; \mathcal{V}(q_i)\; \ldots\; a_n\; \mathcal{V}(q_n)$$

– *For any language $L \subseteq 2^{\mathcal{P}}(\mathcal{A}\,2^{\mathcal{P}})^*$, the transition relation induced by L in $\mathcal{T}$ is defined as $\mathcal{T}(L) = \bigcup_{w \in L} \mathcal{T}(w)$.*

The above definition of programs encompasses the ability (e.g., PDL [15]) to use control constructs such as test-based branching and loops in composing service operations. It is important to note that while we have defined programs and their execution with respect to a single labeled transition system, this is sufficient to describe programs using operations from multiple services. This is because multiple service advertisements can be combined into a single labeled transition system using variants of the asynchronous product construction.

In principle a program $L \in \mathcal{LT}(\mathcal{P}, \mathcal{A})$ can be executed on a transition system $\mathcal{T}$ by non-deterministically selecting and then following the proper trace in L which happens to be an execution trace of $\mathcal{T}$. For practical reasons, we identify a special class of programs that are more desirable to execute, which enjoy two special properties: that of determinism and crash-free execution.

For words x, w, we write $x \preceq w$ to denote that x is a prefix of w, *i.e.*, if there exists a word w' with $w = xw'$. The longest common prefix of two words w_1 and w_2, which we denote by $w_1 \curlywedge w_2$, is the longest word that is a prefix of both w_1 and w_2. We then have the following definition of determinism.

Definition 3 (Determinism). *Two program traces $w_1, w_2 \in 2^{\mathcal{P}}(\mathcal{A}\,2^{\mathcal{P}})^*$ are said to be* consistent*, written $w_1 \sim w_2$, iff $w_1 \neq w_2 \;\; \Rightarrow \;\; w_1 \curlywedge w_2 \in (2^{\mathcal{P}}\,\mathcal{A})^*$. A language $L \subseteq 2^{\mathcal{P}}(\mathcal{A}\,2^{\mathcal{P}})^*$ is* deterministic *iff $\forall w_1, w_2 \in L.\ w_1 \sim w_2$.*

Essentially, two distinct program traces are consistent if, starting from the left, the first position at which they are different is at a propositional valuation. Two consistent program traces are therefore either identical or of the form $w\mathbf{v}_1 w_1$ and $w\mathbf{v}_2 w_2$ with $\mathbf{v}_1 \neq \mathbf{v}_2$.

Let $\mathcal{DLT}(\mathcal{P}, \mathcal{A})$ denote the set of all deterministic languages of program traces over $(\mathcal{P}, \mathcal{A})$. A language $L \in \mathcal{DLT}(\mathcal{P}, \mathcal{A})$ defines executions that are deterministic in a certain sense with respect to an arbitrary (not necessarily deterministic) transition system $\mathcal{T}$, which can be understood informally as follows. If $\mathcal{T}$ is in a state q, we consider all traces of the form $\mathbf{v}w \in L$ whose initial valuation $\mathbf{v}$ matches the valuation in state q. As a consequence of Definition 3, either w is empty or for all such traces the first action a in w is the same. If w is empty, then the execution halts; otherwise, it invokes the action a and continues by executing the program $L' = \{w' \mid \mathbf{v}aw' \in L\}$ (in whatever state $\mathcal{T}$ goes to after the invocation of a). In either case, the next step of either halting or the action a to be invoked and the ensuing program L' to be executed is completely determined by the current state of the transition system. Furthermore, it can be seen that the next program executed $L' \in \mathcal{DLT}(\mathcal{P}, \mathcal{A})$ and therefore the rest of the execution continues to be deterministic.

It is easy to see that the language $L = 2^{\mathcal{P}}a2^{\mathcal{P}} \cup 2^{\mathcal{P}}b2^{\mathcal{P}}$ which corresponds to the regular program $a \cup b$ (for some $a, b \in \mathcal{A}$) does not satisfy Definition 3. Note that if program trace w is a proper prefix of another trace w' then w and w' are not consistent, because after the trace corresponding to w has been followed, w indicates a "halt" while w' indicates an action to invoke. Thus, the language

$L = (2^{\mathcal{P}} a)^* 2^{\mathcal{P}}$ (for some $a \in \mathcal{A}$), which corresponds to the regular program a^*, does not satisfy Definition 3. These two examples illustrate that the general notion of determinism, given by Definition 3, suitably weeds out the intuitively non-deterministic operations of choice and unbounded iteration.

The notion of crash-free execution is most easily motivated through the service advertisement $\mathcal{T}$ given by Figure 1 and the PDL program $l\,;P_l?$ represented by the language $L = 2^{\mathcal{P}}\ l\ \{\mathbf{v} \subseteq 2^{\mathcal{P}} \mid P_l \in \mathbf{v}\}$. In executing L (over $\mathcal{T}$), starting in the initial state q_1, the invocation of the operation l could cause $\mathcal{T}$ to move to the state q_1 where L does not prescribe the next action to perform. At that point, the execution then hangs or crashes. Therefore, although L is deterministic, it is still not a particularly satisfactory program (when executed over $\mathcal{T}$).

More generally, although for a deterministic program $L \in \mathcal{DLT}(\mathcal{P}, \mathcal{A})$, any execution step is always uniquely determined in any state q of a service $\mathcal{T}$, the execution may get "stuck" for one of two reasons: (a) there is no trace whose initial valuation matches the current state q (*i.e.*, there is no next execution step identified), or (b) an action a is invoked but is not enabled in state q (*i.e.*, the next execution step identified cannot be performed). We will filter out such programs by refining the induced transition relation of Definition 2 to identify executions that fail to progress. Define prefix closure $L_{\preceq} \subseteq (2^{\mathcal{P}} \cup \mathcal{A})^*$ as the set of all prefixes of traces in L, *i.e.*, $L_{\preceq} = \{w \mid \exists w' \in L.\ w \preceq w'\}$. The following defines an induced relation with failure $\mathcal{T}^{\pitchfork}(L) \subseteq Q \times (Q \cup \{\pitchfork\})$ where the special symbol $\pitchfork \notin Q$ identifies a failed execution.

Definition 4 (Induced Transition Relation with Failure). *For any $L \in \mathcal{DLT}(\mathcal{P}, \mathcal{A})$ and any transition system $\mathcal{T} = (\mathcal{P}, \mathcal{A}, Q, \mathcal{V}, \{\xrightarrow{a} \mid a \in \mathcal{A}\}, I)$, the relation $\mathcal{T}^{\pitchfork}(L) \subseteq Q \times (Q \cup \{\pitchfork\})$ is defined as follows. For any state $q \in Q$,*

1. *for any $q' \in Q$, $(q, q') \in \mathcal{T}^{\pitchfork}(L)$ iff $(q, q') \in \mathcal{T}(L)$*
2. *$(q, \pitchfork) \in \mathcal{T}^{\pitchfork}(L)$ iff there exists a labeled path $q_0 \xrightarrow{a_1} q_1 \cdots \xrightarrow{a_i} q_i \cdots \xrightarrow{a_n} q_n$ with $n \geq 0$ and $q = q_0$ such that for the word $w = \mathcal{V}(q_0)\ a_1\ \mathcal{V}(q_1)\ \ldots\ a_i\ \mathcal{V}(q_i)\ \ldots\ a_n$ (with w defined to be ϵ if $n = 0$) we have that $w \in L_{\preceq}$ and one of the following conditions holds:*
   - *$w\ \mathcal{V}(q_n) \notin L_{\preceq}$, or*
   - *$w\ \mathcal{V}(q_n)\ a \in L_{\preceq}$ for some $a \in \mathcal{A}$ and $q_n \overset{a}{\nrightarrow}$.*

A deterministic language L is *crash-free* in a state q of a transition system $\mathcal{T}$ if $(q, \pitchfork) \notin \mathcal{T}^{\pitchfork}(L)$, and L is crash-free for a transition system $\mathcal{T}$ if L is crash-free in every state $q \in I$ where I is the set of initial states of $\mathcal{T}$.

3 Query Logic for Specifying Composed Behavior

We now define our logic for specifying the required behavior of the synthesized compositions. Classical modal logics include modalities of the form $\langle a \rangle$ or $[a]$ for action names (that are constants in any model). We generalize these modalities to allow variables so that formulas can include modalities of the form $\langle x \rangle$ or $[x]$. Such variables serve as placeholders for and to identify the different operations of the composed service for which programs need to be synthesized with the overall formula expressing the behavioral property of the execution of these operations, potentially in combination with each other. In this paper, we consider the query logic defined by such an extension to the propositional μ-calculus L_μ [7] since it is among the most expressive specification logics. The solutions to the synthesis problem presented in this paper are applicable to queries specified in any logic (e.g., PDL) translatable to L_μ since the standard translations between modal logics will also apply to their extensions with variable modalities.

The query logic, which we call $L_{\mu,\langle x\rangle}$, is defined as follows.

$$\varphi := p \mid X \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle a\rangle\varphi \mid \langle x\rangle\varphi \mid \mu X\varphi$$

where $p \in \mathcal{P}$ is a propositional constant, $a \in \mathcal{A}$ an action name, $x \in \mathcal{AV}$ is a program variable and $X \in \mathcal{PV}$ is a propositional variable (assuming disjoint sets of variable names $\mathcal{PV}$ and $\mathcal{AV}$). We use lower case letters x, y, x_1 to range over program variables $\mathcal{AV}$ and upper case letters X, Y, X_1 to range over propositional variables $\mathcal{PV}$. The only additional clause in the above grammar to that of L_μ is the formula $\langle x\rangle\varphi$. Similar to L_μ, in the formula $\mu X\varphi$, X is required to appear only positively in φ. Using negation, we can define the propositional constants *true* and *false*, propositional conjunction $\wedge$, propositional implication $\rightarrow$, box modalities $[a]\varphi$ and $[x]\varphi$, as well as greatest fixed point formulas $\nu X\varphi$. Although the syntax of formulas explicitly only allows pure variables to appear within modalities, using the standard translation of PDL [15] into L_μ, we can express formulas of the form $\langle\alpha\rangle\varphi$ and $[\alpha]\varphi$ where α is any regular combination of variables, action names and tests of propositional formulas.

The semantics of a formula is defined over a *LTS* $\mathcal{T}$ to yield a set of states just as in L_μ. However, in addition to an environment η mapping free propositional variables to sets of states, the semantics of $L_{\mu,\langle x\rangle}$ formulas additionally has an environment Θ mapping free program variables to programs in $\mathcal{LT}(\mathcal{P},\mathcal{A})$, so that $\mathcal{T}[\![\varphi]\!]\eta\Theta$ is defined by induction on the structure of φ to return a set of states of $\mathcal{T}$. The inductive case for the variable modality formula is given by

$$\mathcal{T}[\![\langle x\rangle\varphi]\!]\eta\Theta \quad = \quad \{q \mid \exists q'.(q,q') \in \mathcal{T}(\Theta)(x) \text{ and } q' \in \mathcal{T}[\![\varphi]\!]\eta\Theta\}$$

where $\mathcal{T}(\Theta)$ is the map $x \mapsto \mathcal{T}(\Theta(x))$ (see Definition 2 for $\mathcal{T}(\Theta(x))$). For all other forms of formulas, the inductive definition is exactly similar to L_μ [7]. We define satisfaction of a closed formula (i.e., with no free propositional variables) φ as $\mathcal{T} \models \varphi, \Theta$ iff $I \subseteq \mathcal{T}[\![\varphi]\!]\emptyset\Theta$, where I is the set of initial states of $\mathcal{T}$ and $\emptyset$ is the empty environment for propositional variables.

Define an environment Θ to be deterministic and crash-free with respect to a transition system $\mathcal{T}$, if $\Theta(x) \in \mathcal{DLT}(\mathcal{P},\mathcal{A})$ and $\Theta(x)$ is crash-free for $\mathcal{T}$, for every variable x in the domain of Θ. We consider the following synthesis problem.

Problem 1 (Synthesis of Deterministic Crash-Free Programs). Given a labeled transition system $\mathcal{T}$ and a closed $L_{\mu,\langle x\rangle}$ formula φ, does there exist a Θ that is crash-free and deterministic with respect to $\mathcal{T}$ such that $\mathcal{T} \models \varphi, \Theta$? If so, return a Θ that is crash-free and deterministic with respect to $\mathcal{T}$.

Note that the problem only restricts the synthesized programs to be deterministic (in the sense of Definition 3); the transition system $\mathcal{T}$ is allowed to be non-deterministic. As an example, consider the synthesis of a service consisting of two operations: a search function, which will be denoted by the program variable x in the query formula, and a "one-click" checkout function, denoted by the program variable y. Assume that the common ontology includes the propositional variables P_s denoting a successful search and P_c denoting checkout completion as in Figure 1. The property of the search function is that it can always be performed either after a failed search or performing the "one-click" checkout, expressed by the formula $\varphi_{search} \triangleq (\neg P_s \to \langle x\rangle \mathit{true}) \wedge [y]\langle x\rangle \mathit{true}$. Note that even this simple property for x utilizes the ability to use the other program variable y in the formula. The property required of the "one-click" checkout is that it can be performed after a successful search after which checkout completion should always hold, expressed by the formula $\varphi_{checkout} \triangleq P_s \to (\langle y\rangle \mathit{true} \wedge [y]P_c)$. (It is a "one-click" checkout in that its client doesn't have to explicitly do any other operation first such as logging in or adding to the cart.) Finally, we require these properties to hold recursively after any invocation sequence of the synthesized operations (x and y), expressed by the following greatest fixed point formula

$$\varphi \;\triangleq\; \nu X \varphi_{search} \wedge \varphi_{checkout} \wedge [x]X \wedge [y]X$$

We can also express a richer property requiring the use of nested alternate fixed points. Suppose that the synthesized service receives commercial proceeds only from checkouts and being somewhat mercenary, it doesn't want its clients to purely conduct searches without doing checkouts. A more precise formulation capturing this intuitive requirement is that no execution sequence is allowed to contain a subsequence that includes infinitely many successful searches (a search x followed by P_s holding) but without including any checkouts (the operation y). This can be expressed by the formula

$$\varphi_{merc} \;\triangleq\; \mu Y \nu Z [x](P_s \to Y) \wedge [x]Z$$

In understanding the above formula, the $[x]$ in the subformula $[x]Z$ should be read as $[\neg y]$ (*i.e.*, any action other than y which in this example is only the action x). The subformula $[x](P_s \to Y)$ requires that any successful search leads to a state where Y holds; the variable Z then requires this property to hold of the current state and any state reached by a sequence of non-y actions ($[x]Z$). Since the variable Y is bound as a least fixed point, it can only be unwound finitely often. Thus, the formula φ_{merc} holds in states from which every sequence of non-y actions can only include a finite number of x followed by P_s. Following our previous idiom for recursively requiring the property to hold after any execution sequence, the query corresponding to this additional requirement is the formula

$$\psi \;\triangleq\; \nu X \varphi_{search} \wedge \varphi_{checkout} \wedge \varphi_{merc} \wedge [x]X \wedge [y]X$$

4 Solution to the Synthesis Problem

Our solution to the synthesis problem is based on the observation that $\mathcal{T} \models \varphi, \Theta$ depends only on the transition relation induced by Θ, i.e., $\mathcal{T}(\Theta)$. The synthesis algorithm, then, consists of two steps. The first step, presented in Section 4.1, computes constraints on the transition relations induced by the program variable environments which are necessary and sufficient to satisfy a given formula in $L_{\mu,\langle x\rangle}$. The second step, presented in Section 4.2, synthesizes a deterministic crash-free program variable environment whose induced transition relation meets the computed constraints.

4.1 Computing Constraints on Transition Relation

In the first step, we compute necessary and sufficient constraints on the satisfying transition relation assignments for the program variables. Given a *LTS* $\mathcal{T} = (\mathcal{P}, \mathcal{A}, Q, \mathcal{V}, \{\overset{a}{\to} \mid a \in \mathcal{A}\}, I)$ and a closed $L_{\mu,\langle x\rangle}$ formula φ, we are interested in computing the constraint set C such that for any $\hat{\Theta} \in \mathit{Trans}$ we have that $\hat{\Theta} \in C$ iff $\mathcal{T} \models \varphi, \hat{\Theta}$, where *Trans* denotes the function space $\mathcal{AV} \to 2^{Q\times Q}$, *i.e.*, the set of all assignments of program variables to transition relations, and C is a subset of *Trans*. This set C itself cannot be computed compositionally in the formula φ. To permit an inductive definition, we generalize to compute for each state q the constraint set $C(q)$ such that $\hat{\Theta} \in C(q)$ iff $q \in \mathcal{T}[\![\varphi]\!]\hat{\Theta}$.

Following this idea, let *CMap* denote the set of functions $Q \to 2^{\mathit{Trans}}$. Under the pointwise ordering on functions, the set *CMap* is a lattice inheriting the lattice structure of 2^{Trans}. We use $\sqsubseteq$ to denote this ordering and $\bigwedge$ to denote the greatest lower bound. We define a *constraint semantics* $[\![\cdot]\!]^C$ on formulas φ that returns objects in *CMap* with the free variables of φ interpreted similarly, *i.e.*, mapped to objects in *CMap*. Given a $L_{\mu,\langle x\rangle}$ formula φ and an environment $\zeta \in \mathcal{PV} \to \mathit{CMap}$, $\mathcal{T}[\![\varphi]\!]^C\zeta$ as an element of *CMap* is defined inductively:

- $(\mathcal{T}[\![p]\!]^C\zeta)(q) = \mathit{Trans}$ if $p \in \mathcal{V}(q)$ and $\emptyset$ otherwise
- $\mathcal{T}[\![X]\!]^C\zeta = \zeta(X)$
- $(\mathcal{T}[\![\neg\varphi]\!]^C\zeta)(q) = \mathit{Trans}\backslash(\mathcal{T}[\![\varphi]\!]^C\zeta)(q)$
- $(\mathcal{T}[\![\varphi_1 \vee \varphi_2]\!]^C\zeta)(q) = (\mathcal{T}[\![\varphi_1]\!]^C\zeta)(q) \cup (\mathcal{T}[\![\varphi_2]\!]^C\zeta)(q)$
- $(\mathcal{T}[\![\langle a\rangle\varphi]\!]^C\zeta)(q) = \bigcup_{q\overset{a}{\to}q'}(\mathcal{T}[\![\varphi]\!]^C\zeta)(q')$
- $(\mathcal{T}[\![\langle x\rangle\varphi]\!]^C\zeta)(q) = \{\hat{\Theta} \in \mathit{Trans} \mid \exists q' \in Q : (q, q') \in \hat{\Theta}(x), \hat{\Theta} \in (\mathcal{T}[\![\varphi]\!]^C\zeta)(q')\}$
- $\mathcal{T}[\![\mu X\varphi]\!]^C\zeta = \bigwedge\{C \in \mathit{CMap} \mid \mathcal{T}[\![\varphi]\!]^C\zeta[X \mapsto C] \sqsubseteq C\}$, where the environment $\zeta[X \mapsto C]$ maps X to C and otherwise agrees with ζ

Having chosen the appropriate semantic space *CMap*, the inductive semantics is fairly straightforward. Note that the correctness of the semantics of $\neg\varphi$ depends on computing the set of satisfying transition relations *exactly*, and the modal cases highlight the necessity of generalizing to functions on states. The definition of the constraint semantics of $\mu X\varphi$ as the least prefixed point over the space *CMap* (instead of 2^Q as in the standard semantics) is natural but its correctness is somewhat serendipitious. It relies on the following adjunctive structure that can be defined over *CMap*.

Given a $C \in \mathit{CMap}$ and $\hat{\Theta} \in \mathit{Trans}$, define $C_{\upharpoonright\hat{\Theta}} \subseteq Q$ to be the set $\{q \mid \hat{\Theta} \in C(q)\}$, and consider the map $\upharpoonright : \mathit{CMap} \to (\mathit{Trans} \to 2^Q)$ given by $\upharpoonright(C)(\hat{\Theta}) = C_{\upharpoonright\hat{\Theta}}$.

It is easy to see that $\upharpoonright$ preserves least upper bounds and is therefore a left adjoint. Using the commutativity properties of least fixed points with left adjoints (see, *e.g.*, [4]), we can then establish the following lemma, by induction on the structure of the formula, stating that the constraint semantics and standard semantics of a formula are related via this adjunction.

Lemma 1. *For any formula φ, $\zeta \in \mathcal{PV} \to$ CMap, and $\hat{\Theta} \in$ Trans,*

$$(\mathcal{T}[\![\varphi]\!]^C \zeta)_{\upharpoonright\hat{\Theta}} \;=\; \mathcal{T}[\![\varphi]\!]\zeta_{\upharpoonright\hat{\Theta}}\hat{\Theta}$$

From a computational standpoint, a key consequence of Lemma 1 and the adjunction given by $\upharpoonright$ is that the maximum number of iterations required to compute the least fixed point is the height of the lattice 2^Q (*i.e.*, $|Q|$) rather than the height of the lattice *CMap*. We can therefore establish the following complexity for computing the inductive constraint semantics.

Lemma 2. *For any closed formula φ, the time complexity of computing $\mathcal{T}[\![\varphi]\!]^C$ is $O(|\varphi|\;|Q|^{ad(\varphi)+1}\;2^{|Q|^2|\mathcal{AV}|})$, and the space complexity is $O(|Q|^2|\mathcal{AV}|\log(|\varphi|))$, where $ad(\varphi)$ denotes the alternation depth of the formula φ.*

As a corollary of Lemma 1, we can establish the correctness of the constraint semantics for obtaining satisfying transition relations for the program variables.

Corollary 1. *Given φ in $L_{\mu,\langle x\rangle}$ and a substitution Θ, $\mathcal{T} \models \varphi, \Theta$ iff $\mathcal{T}(\Theta)$ is in $(\mathcal{T}[\![\varphi]\!]^C)(q)$, for all $q \in I$.*

4.2 Synthesizing Deterministic Crash-Free Programs

Corollary 1 yields the constraint $C = \bigcap_{q\in I}(\mathcal{T}[\![\varphi]\!]^C)(q)$ as being exactly the set of all satisfying transition relation maps for the program variables, with a program environment Θ being a satisfying solution iff $\mathcal{T}(\Theta) = \hat{\Theta}$ for some $\hat{\Theta} \in C$. The size of C is bounded by $2^{|Q|^2|\mathcal{AV}|}$ and each $\hat{\Theta} \in C$ can be considered in turn. Since $\mathcal{T}(\Theta)$ is given pointwise on each program variable, for any such transition relation map $\hat{\Theta} \in$ *Trans*, a deterministic crash-free program environment Θ (such that $\mathcal{T}(\Theta) = \hat{\Theta}$) can be synthesized for each variable x independently by taking $\Theta(x)$ to be a deterministic crash-free program L such that $\mathcal{T}(L) = \hat{\Theta}(x)$. One is thus reduced to solving the problem of given any transition relation $R \subseteq Q \times Q$, synthesize (if there exists) a deterministic crash-free program X such that $\mathcal{T}(X) = R$. In this section, we present our solution to this problem. Due to space considerations, the presentation is limited to an informal sketch of the main ideas.

Consider any transition relation $R \subseteq Q \times Q$. Define L' to be the language $\bigcap_{(q_1,q_2)\notin R}(2^{\mathcal{P}}(\mathcal{A}\,2^{\mathcal{P}})^* \setminus \mathit{Trace}_T(q_1,q_2))$, and $L'_1,\ldots,L'_{n-1}$ to be an enumeration of the languages $\mathit{Trace}_T(q_1,q_2)$, for every $(q_1,q_2) \in R$. It is easy to show that for any program $X \in \mathcal{LT}(\mathcal{P},\mathcal{A})$ we have that $\mathcal{T}(X) = R$ iff $X \subseteq L'$ and $X \cap L'_i \neq \emptyset$ for $1 \leq i < n$. Let $L_i = L'_i \cap L'$ for $1 \leq i < n$ and $L_n = L'$. We then have that for any program $X \in \mathcal{LT}(\mathcal{P},\mathcal{A})$, $X \subseteq L'$ and $X \cap L'_i \neq \emptyset$ for $1 \leq i < n$ iff

$X \subseteq \cup_i L_i$ and $X \cap L_i \neq \emptyset$ for $1 \leq i \leq n$. Note that the languages $L_1, \dots, L_n$ are regular. Our algorithm to compute a crash-free deterministic program $X \subseteq \cup_i L_i$ which has a non-empty intersection with L_i for $1 \leq i \leq n$, for any given regular languages $L_1, \dots, L_n$, is as follows. It searches for a subautomaton of a certain form in a finite state automaton belonging to a finite set S constructed using the transition system $\mathcal{T}$ and automata $\mathcal{A}_1, \dots, \mathcal{A}_n$ accepting $L_1, \dots, L_n$ respectively. We first present the construction of the set S of finite state automata.

We consider bipartite automata in which the outgoing transitions from one partition have labels in $2^{\mathcal{P}}$ and from the other in $\mathcal{A}$; it is complete if each state has outgoing transitions for all labels in the set corresponding to its partition. The transition system $\mathcal{T}$ can be naturally transformed into a complete bipartite automaton B accepting the trace language of $\mathcal{T}$. Since the languages $L_1, \dots, L_n$ are in $\mathcal{LT}(\mathcal{P}, \mathcal{A})$, there exist deterministic complete bipartite automata $\mathcal{A}_1, \dots, \mathcal{A}_n$ accepting the languages $L_1, \dots, L_n$, respectively. Let $\mathcal{A}'$ be an automaton for $L' = \cup_{1 \leq i \leq n} L_i$ obtained by taking the product of the automata $\mathcal{A}_1, \dots, \mathcal{A}_n$ and choosing appropriate final states. The automata in S are constructed by taking the product of B with an automaton $\mathcal{A}$ constructed by using $\mathcal{A}'$ and trees T, as follows. We consider directed rooted trees T with m nodes where $1 \leq m \leq 2n-1$. Let V be the set of nodes and E the set of edges. Each node is labelled by a distinct number in $\{1, \dots, m\}$ with the root labelled by 1. With each edge (u, v) we associate an edge of $\mathcal{A}'$. Given a tree T as above, we define $Aut(\mathcal{A}', T)$ to be the automaton consisting of a copy of $\mathcal{A}'$ for each node of T, with the following changes in the transitions. If an edge (u, v) of T is labelled by a transition $q \xrightarrow{a} q'$, then instead of the transition $q \xrightarrow{a} q'$ in the i-th copy of A' where i is the label of u, there is a transition from q of the i-th copy on a to q' of the j-th copy where j is the label of v. The set S is the set of all automata obtained by taking the product of B with $Aut(\mathcal{A}', T)$ for some tree T of the above form. Observe that there are only finitely many trees with at most $2n-1$ nodes of the required form, and hence S is finite.

We search for a subautomaton such that the resulting program, taken to be the language of the subautomaton, is deterministic, crash-free and has a non-empty intersection with each L_i. Each of these three requirements translate to the following verifiable conditions on the subautomaton. Assume, without loss of generality, that every state in the subautomaton B' is reachable and can reach a final state of B'. To ensure a non-empty intersection with each L_i, the subautomaton should have for each i, a final state which corresponds to a final state of $\mathcal{A}_i$, and for it to be subset of $\cup_i L_i$, each of its final states should correspond to a final state of $\mathcal{A}_i$ for some i. Similarly, determinism translates to the requirement that from each reachable state in the second partition, there is atmost one action from $\mathcal{A}$ which is enabled. And, for the program to be crash-free with respect to $\mathcal{T}$, for every state in the first partition reachable in the subautomaton which corresponds to a non-dead state q of B, there is a transition out of the state corresponding to the proposition labelling the state q (this ensures that the next execution step is always identified); and for every state in the second partition which corresponds to a non-dead state of B, every transition on an action a out of it leads to a state corresponding to a non-dead state of B (this ensures that the next execution step identified can be performed).

A technical challenge is the proof of correctness, in particular, the proof of the fact that if there exists a deterministic crash-free program P which is a subset of $\cup_i L_i$ and has a non-empty intersection with each L_i, then there exists one which corresponds to the language of a subautomaton of an automaton in S. The proof proceeds by choosing a word from $P \cap L_i$ for each i to form a set W, and using the words in W and the automaton $\mathcal{A}'$ to construct a tree T which is such that there exists a subautomaton of the automaton in S corresponding to this tree whose language is a program of the required form.

The following lemma summarizes the correctness and running time of the synthesis algorithm.

Lemma 3. *Let $\mathcal{T}$ be an LTS over $(\mathcal{P}, \mathcal{A})$ and $L_1, \ldots, L_n$ be regular languages in $\mathcal{LT}(\mathcal{P}, \mathcal{A})$. There is a synthesis algorithm that determines exactly whether there exists a deterministic crash-free program X with respect to $\mathcal{T}$ such that $X \subseteq \bigcup_{1 \leq i \leq n} L_i$ and $X \cap L_i \neq \emptyset$ for $1 \leq i \leq n$ and constructs such an X if there exists one. The time complexity of the algorithm is given by $O(n^n(|\mathcal{T}|$ $(\Pi_i|\mathcal{A}_i|))^n 2^{n|\mathcal{T}|(\Pi_i|\mathcal{A}_i|)})$, where $\mathcal{A}_1, \ldots, \mathcal{A}_n$ are complete bipartite automata for $L_1, \ldots, L_n$ respectively.*

4.3 Example

Consider the service advertisment $\mathcal{T}$ in Figure 1 and the example formula φ from Section 3. The constraint set $((\mathcal{T}[\![\varphi]\!]^C))(q_1)$ includes the following transition relation maps as satisfying solutions:

(1) $x \mapsto \{(q_1, q_1), (q_1, q_3), (q_7, q_2), (q_7, q_4), (q_2, q_2), (q_2, q_4)\}$, $y \mapsto \{(q_3, q_7), (q_4, q_7)\}$
(2) $x \mapsto \{(q_1, q_2), (q_1, q_4), (q_2, q_2), (q_2, q_4), (q_7, q_2), (q_7, q_4)\}$, $y \mapsto \{(q_4, q_7)\}$

The two solutions highlight the interdependence of the transition relations assigned to x and y. In solution (1), x includes the transition (q_1, q_3) (corresponding to not logging in) which forces y to include the transition (q_3, q_7) of checking out from the non logged in state q_3. In solution (2), y does not have to include this transition because x never transitions to the state q_3. Deterministic crash-free programs for the transition relation solutions (1) and (2), respectively, are: (1) $x \mapsto \emptyset s(\emptyset + P_s) + \{P_c\}s(\{P_l\} + \{P_s, P_l\}) + \{P_l\}s(\{P_l\} + \{P_s, P_l\})$, $y \mapsto (\{P_s\}l)^*\{P_s, P_l\}a\{P_a\}c\{P_c\}$ and (2) $x \mapsto (\emptyset l)^+\{P_l\} + \{P_l\}s(\{P_l\} + \{P_s, P_l\}) + \{P_c\}s(\{P_l\} + \{P_s, P_l\})$, $y \mapsto \{P_s, P_l\}a\{P_a\}c\{P_c\}$. In addition to y performing a followed by c, note that the requirement of being crash-free forces any necessary logging in to be performed (by y in Solution (1) and x in Solution(2)) to be realized as, *e.g.*, the language $(\emptyset l)^+\{P_l\}$ rather than the simpler language $\emptyset l\{P_l\}$. The execution of the action login could non-deterministically lead either to successful login or failure, and in either case, the program should identify the next action to be taken. The above two transition relation maps are also satisfying solutions for the example formula ψ since there are no loops in the transition relation for x between states in which search is successful, *i.e.*, containing P_s.

The following transition relation map is also a satisfying solution for φ, but no deterministic crash-free programs can realize the transition relation: $x \mapsto \{(q_1, q_1), (q_1, q_3), (q_1, q_2), (q_2, q_2), (q_2, q_4), (q_7, q_2), (q_7, q_4)\}, y \mapsto \{(q_3, q_7), (q_4, q_7)\}$. No deterministic crash-free program can have both (q_1, q_2) and (q_1, q_3) in its transition relation. Any deterministic program, in state q_1, can chose only one of the two actions s and l, and can consequently stop at at either q_3 or q_4 but not both and cannot include both pairs in its transition relation.

5 Related Work

Semantic Web approaches (e.g., [24], [26]) propose extending ontologies (commonly agreed vocabularies) and using first-order logic to specify behavioral properties of services. Other formalisms have been proposed for specifying the goal application's logic [5, 21, 8, 1, 20]; however, composition is primarily intended to be a manual process.

Most of the work on automatic composition uses AI planning [28, 3] or planning extended to other logics (e.g., Situation Calculus [18, 19], Linear Logic [23]). The main difference from the logic $L_{\mu,\langle x\rangle}$ is that these logics permit the specification of a single synthesized program (as opposed to multiple inter-dependent programs) and specifications are limited to using propositional properties of the state (as opposed to properties of the full execution path).

A closely related theoretical line of work follows what is dubbed the "Roman" model and is represented in [5, 6, 10, 25]. A significant difference from our work is that in the Roman model each of the operations of the synthesized service is mapped to a single action and the synthesized programs therefore correspond to a single invocation of a service operation. On the other hand, the programs considered in this paper are arbitrary languages/automata that can invoke multiple operations sequentially in conjunction with tests, branching, and loops. A second important difference is that in the Roman model, the composed service is explicitly given as a finite state machine whereas we consider its requirements to be specified as a formula in a logic. Apart from our queries therefore having a "declarative" rather than "operational" flavor, a more fundamental consequence is that our queries are inherently more expressive because formulas permit the specification of a set of automata as opposed to a single automaton. As an example, the finiteness requirement property (specified by φ_{merc} or ψ in Section 3) corresponds to an infinite set of automata that cannot be specified as any single finite state machine. On the other hand, requiring trace containment with respect to a specified state machine (the problem addressed in the Roman model) can be encoded in our logic, $L_{\mu,\langle x\rangle}$, as a formula using only one ν and $\langle x\rangle$ modality. A final technical difference is that our synthesis algorithm is more general in that we do not assume that nondeterministic choices in the service advertisements are observationally distinguished in their resulting states.

The problem solved in the second step of our algorithm (Section 4.2) is closely related to the problem of controller synthesis for discrete systems first formulated in [22]. Several technical variants of the problem have been studied; the deterministic requirement on the programs synthesized most closely corresponds to controllers for open systems [17] as opposed to closed systems [14] under a maximal environment. The initial class of synthesis requirements considered [16] were containment in a class of admissible behaviors which corresponds in our setting to requiring certain specified states to be unreachable from the initial states. These results are inapplicable to our problem because the constraints we need to consider additionally specify: (a) states that have to be guaranteed to be reached, and (b) reachability/unreachability conditions starting from arbitrary states (not necessarily initial states). Our problem is most directly reducible to the controller synthesis problem for μ-calculus [2] (CTL^* does not suffice since it cannot express all regular languages). Nevertheless, we chose to directly formulate a new synthesis algorithm because the controller problem for the weaker logic CTL^* is already $3EXPTIME$-complete [17], and a reduction to μ-calculus controller synthesis problem would result in a time-complexity at least one exponential factor worse than given by Lemma 3. The synthesis requirements resulting from $L_{\mu,\langle x\rangle}$, therefore, has an intermediate level of expressiveness that, to the best of our knowledge, has not been addressed previously in the controller synthesis literature.

References

1. OASIS Standard Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, April 2007.
2. A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial observation. *Theoretical Computer Science*, 303(1):7 – 34, 2003.
3. O. Aydin, N. K. Cicekli, and I. Cicekli. Automated web services composition with the event calculus. In *Proc. of 8th International Workshop in Engineering Societies in the Agents World (ESAW07)*, Athens, Greece, 2007.
4. R. C. Backhouse. Galois connections and fixed point calculus. In R. C. Backhouse, R. L. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, pages 89–148. Springer, 2002.
5. D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *Proc. of First International Conference on Service-Oriented Computing (ICSOC 2003)*, pages 43–58, 2003.
6. D. Berardi, M. Mecella, and D. Calvanese. Composing web services with nondeterministic behavior. In *IEEE International Conference on Web Services (ICWS,06*, 2006.
7. J. Bradfield and C. Stirling. Modal mu-calculi. In *Handbook of Modal Logic*, pages 721–756. Elsevier, 2007.
8. F. Casati, S. Ilnicki, L.-J. Jin, V. Krishnamoorthy, and M.-C. Shan. eFlow: A platform for developing and managing composite e-services. In *Proc. of the Academia/Industry Working Conference on Research Challenges (AIWORC '00)*, Washington, DC, USA, 2000.
9. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl, 2001.
10. G. De Giacomo and S. Sardina. Automatic synthesis of new behaviors from a library of available behaviors. In *Proc. of IJCAI-07*, pages 1866–1871, 2007.

11. R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm, 2000.
12. R. Fielding and R. N. Taylor. Principled design of the modern web architecture. In *Proc. of 22nd international conference on Software engineering (ICSE)*, Limerick, Ireland, 2000.
13. M. Hadley. Web application description language (WADL). https://wadl.dev.java.net/wadl20061109.pdf, TR-2006-153, April 2006.
14. S. Jiang and R. Kumar. Supervisory control of discrete event systems with CTL* temporal logic specifications. *SIAM J. Control Optim.*, 44(6):2079–2103, 2006.
15. D. Kozen and J. Tiuryn. Logics of programs. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 789–840. 1990.
16. R. Kumar and V. K. Garg. *Modeling and Control of Logical Discrete Event Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.
17. O. Kupferman, P. Madhusudan, P. Thiagarajan, and M. Y. Vardi. Open systems in reactive environments: Control and synthesis. In *In Proceedings of the 11th International Conference on Concurency Theory, volume 1877 of Lecture Notes in Computer Science*, pages 92–107. Springer-Verlag, 2000.
18. H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3), 1997.
19. S. Mcilraith and T. C. Son. Adapting Golog for composition of semantic Web services. In *Proc. of International Conference on Principles of Knowledge Representation and Reasoning (KR 2002)*, 2002.
20. S. Nanz and T. K. Tolstrup. Goal-oriented composition of services. In *Proc. of the 7th International Symposium on Software Composition (SC'08)*, volume 4954 of *Lecture Notes in Computer Science*. Springer, March 2008.
21. J. Pathak, S. Basu, R. Lutz, and V. Honavar. Selecting and composing web services through iterative reformulation of functional specifications. In *Proc. of 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '06)*, Washington, DC, USA, 2006.
22. P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77, 1989.
23. J. Rao, P. Küngas, and M. Matskin. Composition of semantic web services using linear logic theorem proving. *Information Systems*, 31(4-5):340–360, 2006.
24. D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web service modeling ontology. *Applied Ontology*, 1(1):77–106, 2005.
25. S. Sardina, F. Patrizi, and G. De Giacomo. Automatic synthesis of a global behavior from multiple distributed behaviors. In *AAAI'07: Proceedings of the 22nd national conference on Artificial intelligence*, pages 1063–1069. AAAI Press, 2007.
26. 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.
27. W3C Recommendation. SOAP version 1.2. http://www.w3.org/TR/soap12, April 2007.
28. D. Wu, E. Sirin, J. A. Hendler, D. S. Nau, and B. Parsia. Automatic web services composition using shop2. In *Proc. of World Wide Web Conference*, 2003.