Automatic Composition of Web Services with Nondeterministic Behavior

Daniela Berardi, Giuseppe De Giacomo, Massimo Mecella
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italy
{berardi,degiacomo,mecella}@dis.uniroma1.it

Diego Calvanese
Libera Università di Bolzano/Bozen
Facoltà di Scienze e Tecnologie Informatiche
Piazza Domenicani 3, 39100 Bolzano, Italy
calvanese@inf.unibz.it

ABSTRACT

The promise of Web Service Computing is to utilize Web services as fundamental elements for realizing distributed applications/solutions. In particular, when no available service can satisfy client request, (parts of) available services can be composed and orchestrated in order to satisfy such a request. In this paper, we address the automatic composition when the behavior of the available services is nondeterministic, and hence it is not fully controllable by an orchestrator. The service behavior is modeled by the possible conversations the service can have with its clients. The presence of nondeterministic conversations stems naturally when modeling services in which the result of each interaction with its client can not be foreseen. The behavior of the component services is thus only partially controllable, and an orchestrator needs to cope with such partial controllability. We propose an automatic composition synthesis technique, based on reduction to satisfiability in Propositional Dynamic Logic, that is sound, complete and decidable. Moreover, we will characterize the computational complexity of the problem and show that the proposed technique is optimal wrt computational complexity.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*State diagrams*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of programs*

Keywords

Web Services, Automatic Composition, Nondeterminism, Propositional Dynamic Logic

1. INTRODUCTION

Web services (also called simply services) are self-describing, platform-agnostic computational elements that support rapid, low-cost and easy composition of loosely coupled distributed applications. From a technical standpoint, Web services are modular applications that can be described, published, located, invoked and composed over a variety of networks (including the Internet): any piece of code and any application component deployed on a system can be wrapped and transformed into a network-available service, by using standard (XML-based) languages and protocols (e.g., WSDL, SOAP, etc.) - see e.g., [1].

The promise of Web service is to enable the composition of new distributed applications/solutions: when no available service can satisfy a client request, (parts of) available services can be composed and orchestrated in order to satisfy such a request. Note that service composition involves two different issues [1]: the *synthesis*, in order to synthesize, either manually or automatically, a specification of how coordinating the component services to fulfill the client request, and the *orchestration*, i.e., how executing the previous obtained specification by suitably supervising and monitoring both the control flow and the data flow among the involved services.

In this paper, we address the automatic composition synthesis when the behavior of the available services is nondeterministic, and hence is not fully controllable by the orchestrator. The service behavior is modeled by the possible conversations the service can have with its clients. The presence of nondeterministic conversations stems naturally when modeling services in which the result of each interaction with its client on the state of the service can not be foreseen. Let us consider as an example, a service that allows buying items by credit card; after invoking the operation, the service can be in a state `payment_OK`, accepting the payment, or in a different state `payment_refused`, if the credit card is not valid, with not enough credit, etc. Note that the client of a nondeterministic service can invoke the operation but cannot control what is the result of it. In other words, the behavior of the service is partially controllable, and the orchestrator needs to cope with such partial controllability. Note also that if one observes the status in which the service is after an operation, then s/he understand which transition, among those nondeterministically possible in the previous state, has been undertaken by the service. We assume that the orchestrator can indeed observe states of the available services and take advantage of this in choosing how to continue a certain task [1].

From a formal point of view, in this paper, we adhere to the setting proposed in [3, 4, 5] whose distinguished features can be summarized as follows.

- The available services are grouped together into a so call *community*.
- Services in the community share a common set of actions Σ, the *actions of the community*. In other words, each available

Univ. Roma "La Sapienza", Dipartimento di Informatica e Sistemistica. Technical Report 05-2006.

[1]The reader should observe that also the standard proposal WSDL 2.0 has a similar point of view: the same operation can have multiple output messages (the `out message` and various `outfault messages`), and the client observe how the service behaved only after receiving a specific output message.

service in the community exports its behavior to the community itself in terms of the actions in Σ (the actions recognized be the community).

- Each action in Σ denotes a (possibly complex) interaction between the a service and a client, and as a result of such interaction the client may acquire new information (not necessarily modeled explicitly) that may be of help in choosing the next action to perform.

- The behavior of each available service is described in terms of a *finite transition system* (aka finite state machine) that makes use of the actions in Σ. Since in this paper we assume that the behavior of the available services is nondeterministic, differently form [3, 4, 5], such a transition system are nondeterministic in general.

- The client request itself is expressed as a finite transition system that makes use of the actions in Σ. Such a transition system, called *target service*, is deterministic as in [3, 5], since we assume that there is no uncertainty on the behavior that the client want to realize through composition of the available services.

- The orchestrator has the ability of scheduling services on a *step-by-step* basis. Hence the orchestrator has the ability of *controlling the interleaving* of multiple services executed concurrently.

- The *composition synthesis* consists on synthesizing a program for the orchestrator such that by suitably scheduling the available services it can provide the target service to the client.

The contribution of this paper is to devise a formal technique to perform automatic composition synthesis, when available services are nondeterministic and hence partially controllable by the orchestrator. We will show that the technique proposed is sound, complete and terminating. Moreover we will characterize the computational complexity of the problem and show that the proposed technique is optimal wrt (worst-case) computational complexity.

Typically reactive process synthesis [23, 15] make use of techniques based on automata on infinite trees. And even if these are perfectly suitable from a theoretical point of view, there are critical steps, such a Safra's construction for complementation, that have resisted efficient implementation for a long time. Only very recently, we are starting to understand how to avoid such steps – see [14] for a discussion.

Interestingly the technique proposed here is based on reduction to satisfiability in Propositional Dynamic Logic (PDL) [12] with a limited use of the reflexive-transitive-closure operator[2]. Now, PDL satisfiability shares the same basic algorithms behind the success of the description logics-based reasoning systems used for OWL[3], such as FaCT[4], Racer[5], Pellet[6], and hence its applicability in the context of composition synthesis appears to be quite promising.

The rest of the paper is organized as follows. In Section 2 we first introduce the setting and the composition problem in formal details. In Section 3 we develop the techniques to perform automatic composition, we show soundness and completeness and we characterize the complexity of both the techniques and the problem. In Section 4 we study an extension of the setting where transitions in the available services and in the target service can be guarded by conditions on some shared variables. The main objective of this extension is to show that the techniques proposed in Section 3 are actually quite resistant to significant variation of the setting (for another example see [2]). In Section 5 we discuss some related work. Finally, in Section 6 we draw some conclusions.

2. SERVICES WITH PARTIALLY CONTROLLABLE BEHAVIOR

In this section, we formalize composition when the services that are available in the community have a behavior that is not fully controllable by the orchestrator.

Formally, we consider each *available service* as a *nondeterministic*[7] finite transition system $\mathcal{S} = (\Sigma, S, s_0, \delta, F)$ where Σ is the common alphabet of actions of the community, S is a finite set of states, $s_0 \in S$ is the single initial state, $\delta \subseteq S \times \Sigma \times S$ is the transition relation[8], and $F \subseteq S$ is the set of final states, that is, states in which the computation may stop, but does not necessarily have to.

The client service request is expressed as a *target service*, which represents the service the client would like to interact with. Such a service is again modeled as a finite transition system over the alphabet of the community, but this time a *deterministic* one, i.e., the transition relation is actually functional (there cannot be two distinct transitions with the same starting state and action). The target service is deterministic because we assume that the client has full control on how to execute the service that he/she requires[9].

EXAMPLE 2.1. *Figure 1(a) shows a community of services for getting information on books. The community includes two services: $\mathcal{S}_1$ that allows one to repeatedly* (i) *search the ISBN of a book given its title (`search`) then,* (ii) *in certain cases (e.g., if the record with cataloging data is currently accessible), it allows for displaying the cataloging data (such as editor information, year of publication, authors, copyrights, etc.) of the book with the selected ISBN (`display`), or* (iii) *simply returns without displaying information (`return`); $\mathcal{S}_2$ allows for repeatedly displaying cataloging data of books given the ISBN (`display`), without allowing researches. Figure 1(b) shows the target service $\mathcal{S}_0$: the client wants to have a service that allows him to search for a book ISBN given its title (`search`), and then display its cataloging data (`display`). Note that the client wants to display the cataloging data in any case and hence he/she can neither directly exploit $\mathcal{S}_1$ nor $\mathcal{S}_2$.*

Next, we need to clarify which are the basic capabilities of the orchestrator. The orchestrator has the ability of selecting one[10] of the available services and instructing it to execute an action among those available in its current state. Furthermore, the orchestrator has the ability of keeping track (at runtime) of the current state of

[2]As in [3, 5], but more sophisticated this time in order to correctly deal with nondeterministic behavior of the available services.

[3]`http://www.omg.org/uml/`

[4]`http://www.cs.man.ac.uk/~horrocks/FaCT/`

[5]`http://www.sts.tu-harburg.de/~r.f.moeller/racer/`

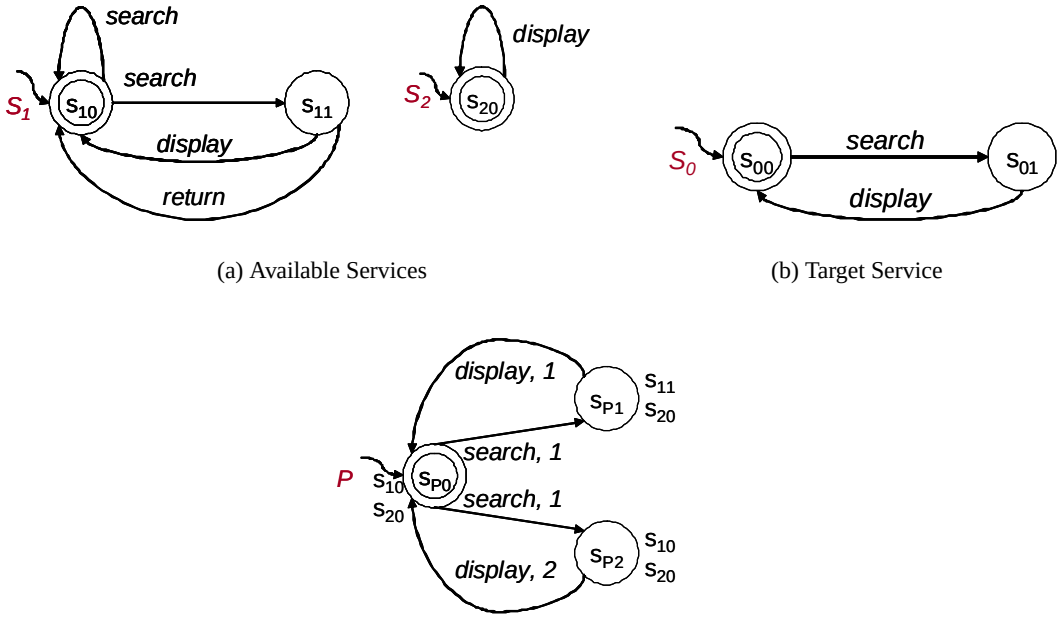
[6]`http://www.mindswap.org/2003/pellet/`

[7]Note that this kind of nondeterminism is of a *devilish* nature, so as to capture the idea that the orchestrator cannot fully control the available services.

[8]As usual, we call the Σ component of such triples, the *label of the transition*.

[9]In fact we could have a client request that is expressed as a nondeterministic transition system as in [4]. In this case, however, the nondeterminism has a *don't-care*, aka *angelic* nature.

[10]For simplicity, we assume that the orchestrator selects only one service at each step, however our approach and results easily extend to the case where more services can be selected at each step.



(a) Available Services

(b) Target Service

(c) Composition

Figure 1: Composition of nondeterministic services

each available service. Technically such a capability is called *full observability* on the states of the available services. Although other choices are possible [26, 2], full observability is the natural choice in this context, since the transition system that each available service exposes to the community is specific to the community itself (indeed it is expressed using the common alphabet of actions of the community), and hence there is no reason to make its states partially unobservable: if details have to be hidden, this can be done directly within the transition system, possibly making use of nondeterminism.

We are now ready to define composition synthesis: an "orchestrator program"[11] that the orchestrator has to execute in order to orchestrate the available services in order to offer the target service to the client. Let the available service be $\mathcal{S}_1, \ldots, \mathcal{S}_n$ each with $\mathcal{S}_i = (\Sigma, S_i, s_{i0}, \delta_i, F_i)$, and the target service $\mathcal{S}_0 = (\Sigma, S_0, s_{00}, \delta_0, F_0)$. A *history* is an alternating sequence of the form $h = (s_1^0, \ldots, s_n^0) \cdot a^1 \cdot (s_1^1, \ldots, s_n^1) \cdots a^\ell \cdot (s_1^\ell, \ldots, s_n^\ell)$ such that the following constraints hold:

- $s_i^0 = s_{i0}$ for $i \in \{1, \ldots, n\}$, i.e., all services start in their initial state;
- at each step k, for one i we have that $(s_i^k, a^{k+1}, s_i^{k+1}) \in \delta_i$, while for all $j \neq i$ we have that $s_j^{k+1} = s_j^k$, i.e., at each step of the history, only one of the service has made a transition (according to its transition relation), while the other ones have remained still.

An *orchestrator program* is a function $P : \mathcal{H} \times \Sigma \to \{1, \ldots, n, u\}$ that, given a history $h \in \mathcal{H}$ (where $\mathcal{H}$ is the set of all histories defined as above) and an action $a \in \Sigma$ to perform, returns the service (actually the service index) that will perform it. Observe that such a function may also return a special value u (for "undefined"). This is a technical convenience to make P a total function returning values even for histories that are not of interest or for actions that no service can perform after a given history.

Next, we define when an orchestrator program is a composition that realizes the target services. First, we observe that, since the target service is a deterministic transition system, its behavior is completely characterized by the set of its traces, i.e., by the set of infinite sequences of actions that are faithful to its transitions, and of finite sequences that in addition lead to a final state[12]. Now, given a trace $t = a_1 \cdot a_2 \cdots$ of the target service, we say that an *orchestrator program P realizes the trace t* iff for each non-negative integer ℓ and for each history $h \in \mathcal{H}_t^\ell$, we have that $P(h, a_{\ell+1}) \neq u$ and $\mathcal{H}_t^{\ell+1}$ is nonempty, where the sets $\mathcal{H}_t^\ell$ are inductively defined as follows:

- $\mathcal{H}_t^0 = \{(s_{10}, \ldots, s_{n0})\}$
- $\mathcal{H}_t^{\ell+1}$ is the set of all histories such that, if $h \in \mathcal{H}_t^\ell$ and $P(h, a_{\ell+1}) = i$ (with $i \neq u$), then for all transitions $(s_i^\ell, a, s_i') \in \delta_i$ the history $h \cdot a_{\ell+1} \cdot (s_1^{\ell+1}, \ldots, s_n^{\ell+1})$, with $s_i^{\ell+1} = s_i'$, and $s_j^{\ell+1} = s_j^\ell$ for $j \neq i$, is in $\mathcal{H}_t^{\ell+1}$.

Moreover, if a trace is finite and ends after f actions, we have that all histories in $\mathcal{H}_t^f$ end with all services in a final state. Finally, we say that an *orchestrator program P realizes the target service $\mathcal{S}_0$*, if it realizes all its traces.

In order to understand the above definitions, let us observe that intuitively the orchestrator program realizes a trace if it can choose

[11] In fact, this is a skeleton specification of the actual program for the orchestrator.

[12] Actually, the behavior captured by a transition system is typically identified with its execution tree, see [3]. However, since the target service has a deterministic transition system, the set of traces is sufficient, since one can immediately reconstruct the execution tree from it.

at every step an available service to perform the requested action. However, since when an available service executes an action it nondeterministically chooses what transition to actually perform, the orchestrator program has to play on the safe side and require that for each of the possible resulting states of the activated service, the orchestrator is able to continue with the execution of the next action. In addition, before ending a computation, available services need to be left in a final state, hence we have the additional requirement above for finite traces.

EXAMPLE 2.2. *Referring to Example 2.1, Figure 1(c) shows an orchestrator program P (in this case with finite states) for available services $\mathcal{S}_1$ and $\mathcal{S}_2$ in Figure 1(a), that realizes the target service $\mathcal{S}_0$ in Figure 1(b). Essentially, P behaves as follows: it repeatedly delegates to $\mathcal{S}_1$ the action* `search` *(notice that both transitions labeled with this actions are delegated to $\mathcal{S}_1$); then it checks the resulting state of $\mathcal{S}_1$ and, depending on this state, it delegates the action* `display` *to either $\mathcal{S}_1$ or $\mathcal{S}_2$.*

Observe that, in general, an orchestrator program could require infinite states. However, we will show next that if an orchestrator program that realizes the target service exists, then there exists one with finite states. Note that, even if it has finite states, the orchestrator program has to observe the states of the available services in order to decide which service to select next (for a given action requested by the target service). This makes such orchestrator programs akin to an advanced form of conditional plans studied in AI [24].

3. COMPOSITION SYNTHESIS WITH PARTIAL CONTROLLABILITY

In this section, we study how to check for the existence of an orchestrator program that realizes the target service, and actually computing it. We start by some preliminary considerations on the computational complexity that we should expect. Muscholl and Walukiewicz in [20] proved the following lowerbound.

THEOREM 3.1 (MUSCHOLL&WALUKIEWICZ 2005). *Checking the existence of an orchestrator program for available services $\mathcal{S}_1, \ldots, \mathcal{S}_n$ that realizes a target service $\mathcal{S}_0$ is EXPTIME-hard*[13].

So we should expect that checking the existence of a composition is at least exponential time. Here we show that actually the problem is EXPTIME-complete, by resorting to a reduction to satisfiability in Propositional Dynamic Logic (PDL). Moreover, such a reduction can be exploited to generate the actual orchestrator program realizing the composition, which is finite. In doing this, we extend the approach in [3, 4, 5] to deal with nondeterministic available services. Dealing with such nondeterminism requires solving same subtle points that reflect the sophisticated notion of orchestrator program needed for that.

3.1 Propositional Dynamic Logic

Propositional Dynamic Logic (PDL) is a modal logic specifically developed for reasoning about computer programs [12]. Syntactically, PDL formulas are built by starting from a set $\mathcal{P}$ of atomic propositions and a set Σ of atomic actions as follows:

$$\begin{array}{rcl} \phi & \longrightarrow & P \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi \rightarrow \phi' \mid \\ & & \langle r \rangle \phi \mid [r]\phi \mid \texttt{true} \mid \texttt{false} \\ r & \longrightarrow & a \mid r_1 \cup r_2 \mid r_1; r_2 \mid r^* \mid \phi? \end{array}$$

where P is an atomic proposition in $\mathcal{P}$, r is a regular expression over the set of actions in Σ, and a is an atomic action in Σ. That is, PDL formulas are composed from atomic propositions by applying arbitrary propositional connectives, and modal operators $\langle r \rangle \phi$ and $[r]\phi$. The meaning of the modal operators is, respectively, that there exists an execution of r reaching a state where ϕ holds, and that all terminating executions of r reach a state where ϕ holds. As far as programs, $r_1 \cup r_2$ means "choose non deterministically between r_1 and r_2"; $r_1; r_2$ means "first execute r_1 then execute r_2"; r^* means "execute r a non deterministically chosen number of times (zero or more)"; $\phi?$ means "test ϕ: if it is true proceed else fail".

Among the programs, $(\cup_{a \in \Sigma} a)^*$, abbreviated as $\mathbf{u}$, has a special importance: $[\mathbf{u}]$ represents the *master modality*, which can be used to state universal assertions. Indeed, $\mathbf{u}$ is defined as the reflexive and transitive union of all actions in Σ and represents the iteration of a nondeterministic choice among all the possible atomic programs [12].

The semantics of PDL formulas is based on the notion of Kripke structure. A Kripke structure is a triple of the form $\mathcal{I} = (\Delta^{\mathcal{I}}, \{a^{\mathcal{I}}\}_{a \in \Sigma}, \{P^{\mathcal{I}}\}_{P \in \mathcal{P}})$, where $\Delta^{\mathcal{I}}$ denotes a non-empty set of states (also called worlds); $\{a^{\mathcal{I}}\}_{a \in \Sigma}$ is a family of binary relations $a^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ between elements of $\Delta^{\mathcal{I}}$, each of which denotes the state transitions caused by the atomic program a; and $\{P^{\mathcal{I}}\}_{P \in \mathcal{P}}$ is a family of unary relations $P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ each of which denotes the elements of $\Delta^{\mathcal{I}}$ were the proposition P is true. The semantic relation "a formula ϕ holds at a state s of a structure $\mathcal{I}$", written $\mathcal{I}, s \models \phi$, is defined by induction on the form of ϕ:

$\mathcal{I}, s \models \texttt{true}$	always	
$\mathcal{I}, s \models \texttt{false}$	never	
$\mathcal{I}, s \models P$	iff	$s \in P^{\mathcal{I}}$
$\mathcal{I}, s \models \neg\phi$	iff	$\mathcal{I}, s \not\models \phi$
$\mathcal{I}, s \models \phi_1 \wedge \phi_2$	iff	$\mathcal{I}, s \models \phi_1$ and $\mathcal{I}, s \models \phi_2$
$\mathcal{I}, s \models \phi_1 \vee \phi_2$	iff	$\mathcal{I}, s \models \phi_1$ or $\mathcal{I}, s \models \phi_2$
$\mathcal{I}, s \models \phi \rightarrow \phi'$	iff	$\mathcal{I}, s \models \phi$ implies $\mathcal{I}, s \models \phi'$
$\mathcal{I}, s \models \langle r \rangle \phi$	iff	exists s' s.t. $(s, s') \in r^{\mathcal{I}}$ and $\mathcal{I}, s' \models \phi$
$\mathcal{I}, s \models [r]\phi$	iff	for all s', $(s, s') \in r^{\mathcal{I}}$ implies $\mathcal{I}, s' \models \phi$

where the family $\{a^{\mathcal{I}}\}_{a \in \Sigma}$ is systematically extended so as to include, for every program r, the corresponding function $r^{\mathcal{I}}$ defined by induction on the form of r:

$$\begin{array}{lcl} a^{\mathcal{I}} : & \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} & \\ (r_1 \cup r_2)^{\mathcal{I}} & = & r_1^{\mathcal{I}} \cup r_2^{\mathcal{I}} \\ (r_1; r_2)^{\mathcal{I}} & = & r_1^{\mathcal{I}}; r_2^{\mathcal{I}} \\ (r^*)^{\mathcal{I}} & = & (r^{\mathcal{I}})^* \\ (\phi?)^{\mathcal{I}} & = & \{(s, s) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid \mathcal{I}, s \models \phi\} \end{array}$$

A structure $\mathcal{I} = (\Delta^{\mathcal{I}}, \{a^{\mathcal{I}}\}_{a \in \Sigma}, \{P^{\mathcal{I}}\}_{P \in \mathcal{P}})$ is called a *model* of a formula ϕ if there exists a state $s \in \Delta^{\mathcal{I}}$ such that $\mathcal{I}, s \models \phi$. A formula ϕ is *satisfiable* if there exists a model of ϕ, otherwise the formula is *unsatisfiable*.

Next two theorems give us the complexity characterization of satisfiability in PDL.

THEOREM 3.2 (FISHER&LADNER 1977). *Satisfiability in PDL is EXPTIME-hard.*

THEOREM 3.3 (PRATT 1978). *Satisfiability in PDL is EXPTIME-complete.*

[13] In fact Muscholl and Walukiewicz show EXPTIME-hardness for the simpler setting, studied in [3, 5], where all available services are deterministic [20].

PDL enjoys two properties that are of particular interest [12], which we exploit in our composition technique. The first is the *tree model property*, which says that every model of a formula can be unwound to a (possibly infinite) tree-shaped model (considering domain elements as nodes and partial functions interpreting actions as edges). The second is the *small model property*, which says that every satisfiable formula admits a finite model whose size (in particular the number of domain elements) is at most exponential in the size of the formula itself.

3.2 Reduction to PDL

Given the specification of a target service $\mathcal{S}_0 = (\Sigma, S_0, s_{00}, \delta_0, F_0)$ and available services $\mathcal{S}_i = (\Sigma, S_i, s_{0i}, \delta_i F_i)$, for $i \in \{1, \ldots, n\}$, we build a PDL formula Φ to check for satisfiability as follows.

As actions in Φ we have the actions of the community Σ, and as atomic propositions we have: *(i)* one atomic proposition s for each $i \in \{0, 1, \ldots, n\}$ and each state s of $\mathcal{S}_i$, which intuitively denotes that $\mathcal{S}_i$ is in state s[14]; *(ii)* atomic propositions F_i, for $i \in \{0, 1, \ldots n\}$, denoting that S_i is in a final state; *(iii)* atomic propositions $exec_{ia}$, for $i \in \{1, \ldots n\}$ and $a \in \Sigma$, denoting that a will be executed next by the available service $\mathcal{S}_i$; *(iv)* one atomic proposition $undef$ denoting that we reached a situation where the orchestrator program can be left undefined.

The formula Φ is formed as follows. For representing the transitions of the target service $\mathcal{S}_0$ we construct a formula ϕ_0 as the conjunction of:

- $s \rightarrow \langle a \rangle \mathtt{true} \wedge [a]s'$, for each transition $(s, a, s') \in \delta_0$, encoding that the target service can do an a-transition going from state s to state s'.
- $s \rightarrow [a]undef$, for each $s \in S_0$ and action a such that for no s' we have $(s, a, s') \in \delta_0$, which takes into account that the target service cannot perform an a-transition.

For representing the transitions of each available service $\mathcal{S}_i$, we construct a formula ϕ_i as the conjunction of:

- $s \wedge exec_{ia} \rightarrow \langle a \rangle s'_1 \wedge \cdots \wedge \langle a \rangle s'_m \wedge [a](s'_1 \vee \cdots \vee s'_m)$, where $\{s'_1, \ldots, s'_m\} = \{s' \mid (s, a, s') \in \delta_i\}$, for each $s \in S_i$ and each action a such that $(s, a, s') \in \delta_i$ for some s'; these assertions encode that if a service $\mathcal{S}_i$ is in state s and is selected for the execution of an action a (i.e., $exec_{ia}$ is true), then for each possible a-transition, we have a possible a-successor in the models of Φ.
- $s \wedge exec_{ia} \rightarrow [a]\mathtt{false}$, for each $s \in S_i$ such that for no s' we have that $(s, a, s') \in \delta_i$; these denote that if service $\mathcal{S}_i$ is in state s, is selected for the execution of a, and it cannot do a, then there is no a-successor in the models of Φ.
- $s \wedge \neg exec_{ia} \rightarrow [a]s$ for each state $s \in S_i$ and each action a; encoding that if service $\mathcal{S}_i$ is in state s and is not selected for the execution of a, then if a is performed (by some other available service), $\mathcal{S}_i$ does not change state.

In addition we have the formula ϕ_{add} formed as the conjunction of:

- $s \rightarrow \neg s'$, for all pairs of states $s, s' \in \mathcal{S}_i$, and for $i \in \{0, 1, , \ldots, n\}$; these say that propositions representing different states of $\mathcal{S}_i$ are disjoint.
- $F_i \leftrightarrow \bigvee_{s \in F_i} s$, for $i \in \{0, 1, , \ldots, n\}$; this highlights final states of $\mathcal{S}_i$.
- $undef \rightarrow [a]undef$, for each action $a \in \Sigma$; these say that once a situation is reached where $undef$ holds, then $undef$ holds also in all successor situations.
- $\neg undef \wedge \langle a \rangle \mathtt{true} \rightarrow \bigvee_{i \in \{1,\ldots,n\}} exec_{ia}$, for each $a \in \Sigma$, denoting that, unless $undef$ is true, if a is performed, then at least one of the available services must be selected for the execution of a.
- $exec_{ia} \rightarrow \neg exec_{ja}$ for each $i, j \in \{1, \ldots, n\}$, $i \neq j$, and each $a \in \Sigma$, stating that only one available service is selected for the execution of a.
- $F_0 \rightarrow \bigwedge_{i \in \{1,\ldots,n\}} F_i$; this says that when the target service is in a final state also all available services must be in a final state.

Finally, we define Φ as $Init \wedge [\mathbf{u}](\phi_0 \wedge \bigwedge_{i \in \{1,\ldots,n\}} \phi_i \wedge \phi_{add})$, where $Init$ stands for $s_{00} \wedge s_{01} \wedge \cdots \wedge s_{0n}$, and represents the initial states of all services, and $\mathbf{u} = (\bigcup_{a \in \Sigma} a)^*$ is the master modality, which is used to force $\phi_0 \wedge \bigwedge_{i \in \{1,\ldots,n\}} \phi_i \wedge \phi_{add}$ to be true in every point of the model. Note that $\mathbf{u}$ is the only complex program that appears in the PDL formula Φ. We can now state our main result.

THEOREM 3.4. *The PDL formula Φ, constructed as above, is satisfiable if and only if there exists an orchestrator program for the available services $\mathcal{S}_1, \ldots, \mathcal{S}_n$ that realizes the target service $\mathcal{S}_0$.*

Proof (sketch). "If": PDL has the tree-model property. Hence, if Φ is satisfiable then it has a model that is tree shaped. Each node in this tree can be put in correspondence with a history, and from the truth value assignment of the propositions $exec_{ia}$ in the node one can reconstruct the orchestrator program. *"Only if"*: if an orchestrator program that realizes $\mathcal{S}_0$ exists, one can use it to build a tree model of Φ. □

Observe that the size of Φ is polynomially related to $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_n$. Hence, from the EXPTIME-completeness of satisfiability in PDL and Theorem 3.1 we get:

THEOREM 3.5. *Checking the existence of an orchestrator program for the available services $\mathcal{S}_1, \ldots, \mathcal{S}_n$ that realizes the target service $\mathcal{S}_0$ is EXPTIME-complete.*

Finally by the finite-model property of PDL, i.e., if a formula is satisfiable it is satisfiable in a model that is at most exponential in the size of the formula, we get a systematic procedure for synthesizing the composition:

THEOREM 3.6. *If there exists an orchestrator program for the available services $\mathcal{S}_1, \ldots, \mathcal{S}_n$ that realizes the target service $\mathcal{S}_0$, then there exists one that requires a finite number of states. Moreover such a finite state program can be extracted from a finite model of Φ.*

[14] In this paper we are not concerned with compact representations of the states of S_i. However, we observe that if states are succinctly represented (e.g., in binary format) then, in general, we can exploit in Φ such a representation to get a corresponding compact formula as well.

4. A POSSIBLE EXTENSION: GUARDED SERVICES

Our approach to deal with nondeterministic available services can be extended in several directions. Here we study an extension based on the introduction of a set of *variables shared among the available services and the client* that encode some basic information that is exchanged between the services, and that the client acquires while executing the target service. Observe that by no means we should think of such variables as a full representation of the client knowledge; indeed, we assume that much more information is passed to the client when performing actions and such information is used by the client to actually select what action to take next. Once we introduce shared variables, we can use them to guard transitions in both the target and the available services. Moreover we have to model how available services change the values of variables when executing actions.

Formally, now the community shares, in addition to the common alphabet of actions Σ, also a common alphabet $\mathcal{V}$ of atomic variables. Each such a variable can assume values from a finite set Δ. To be more concrete, in our examples we will consider as possible values "known to be true" denoted by tt, "known to be false" denoted by ff, and "not known", denoted by uu. We denote by Ψ the set of propositional formulas whose atoms are equalities (interpreted in the obvious way) involving variables and values. We also denote by Γ the set of (possibly partial) assignments of values in Δ to the variables in $\mathcal{V}$. Available services are defined in terms of finite transition systems as before, except that this time the action alphabet used to label the transitions is not Σ, but $\Psi \times \Sigma \times \Gamma$. That is, now transitions are labeled by an action a that causes the transition, a guard g that must hold given the current variable assignment in order to perform the transition, and a possibly partial reassignment c of the variables in $\mathcal{V}$ resulting after the transitions. We denote such labels as $\{g\}a\{c\}$. We will drop the guard g to mean that $g = true$, and drop the reassignment if c does not assign any variable.

EXAMPLE 4.1. *Figure 2(a) shows a community of services for buying selected items with prepaid cards. $\mathcal{S}_1$ allows to select and add the items (at least one) to the cart, and then, if the remaining amount of money in the card is known to be enough (i.e.,* `Enough=tt`*), the client can buy the item(s), otherwise, if the money availability is known not to be enough (i.e.,* `Enough=ff`*) or it is unknown (*`Enough=uu`*), the client can choose to either abort the service or retry the purchase. $\mathcal{S}_2$ allows to fill in the prepaid card: note that the amount of money after the execution of operation* `fill` *is always known to be enough, independently of whether, before its execution, it was known to be enough, known to be not enough or unknown. Finally, $\mathcal{S}_3$ allows for checking the amount of money on the prepaid card: if its value is known to be enough (not enough, resp.), then* `checkPP` *simply returns* `Enough=tt` *(*`Enough=ff`*, resp.); if its value is unknown, then* `checkPP` nondeterministically *returns either* `Enough=tt` *or* `Enough=ff`*.*

The client specification now is formed by two components:

- a target service, defined as a deterministic finite transition system, as before, but where transitions are labeled by $\Psi \times \Sigma$ instead of just Σ, i.e., now actions causing the transitions are guarded; observe that the notion of determinism can be slightly changed: there cannot be two transitions with the same action a and with guards g and g' that are not mutually exclusive (i.e., $g \wedge g' = false$ for every possible assignment of values in Δ to the variables in $\mathcal{V}$);
- an *initial assignment* VA_{init} to the variables in $\mathcal{V}$, describing the circumstances in which the client wants the target service to work.

EXAMPLE 4.2. *Referring to Example 4.1, Figure 2(b) shows a target service $\mathcal{S}_0$ specifying that, after selecting and adding the items to the cart, if the money on the prepaid card is known to be enough, the client wants to buy them, otherwise, he/she wants to first fill in the card and then buy the items. The same figure also shows initial assignment to the single shared variable* `Enough`*, whose value is initially unknown.*

We are ready to define composition. A *history* is an alternating sequence of the form $h = (s_1^0, \ldots, s_n^0, VA^0) \cdot a^1 \cdot (s_1^1, \ldots, s_n^1, VA^1) \cdots a^\ell \cdot (s_1^\ell, \ldots s_n^\ell, VA^\ell)$ such that the following constraints hold:

- $s_i^0 = s_{i0}$ for $i \in \{1, \ldots, n\}$, i.e., all services start in their initial state, and VA^0 is a total assignment of all variables in $\mathcal{V}$;
- at each step k, *(i)* for one i we have that $(s^k, \{g\}a^{k+1}\{c\}, s_i^{k+1}) \in \delta_i$ with $g = true$ in VA^k; *(ii)* $VA^{k+1} = VA^k; c$ where $VA^k; c$ stands for the assignment obtained from VA^k by reassigning the variables mentioned in c according to c itself; *(iii)* for all $j \neq i$ we have that $s_j^{k+1} = s_j^k$.

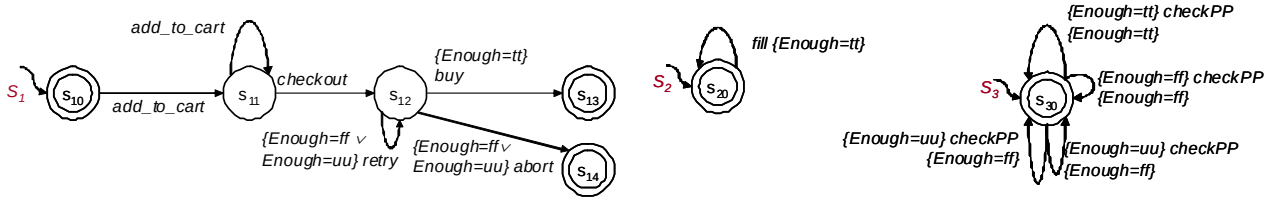
An *orchestrator program* is a function $P : \mathcal{H} \times \Sigma \to \{1, \ldots, n, u\}$ that, given a history $h \in \mathcal{H}$ (where $\mathcal{H}$ is the set of all histories defined as above) and an action $a \in \Sigma$ to perform, returns the service (actually the service index) that will perform it, or the special value u (for "undefined").

Next we define when an orchestrator program is a composition that realizes the client request. Again, being the target service deterministic, its behavior is completely characterized by the set of its traces, this time defined by the set of infinite sequences of *guarded* actions that are faithful to its transitions, and of finite sequences that in addition lead to a final state. Now, given a trace $t = (g^1, a_1) \cdot (g^2, a_2) \cdots$ of the target service, we say that an *orchestrator program P realizes the trace t starting from an initial variable assignment VA_{init}* iff for all ℓ and for all histories in $h \in \mathcal{H}_t^\ell$ such that $g^{\ell+1} = true$ in the last variable assignment VA_h^ℓ of h, we have that $P(h, a_{\ell+1}) \neq u$ and $\mathcal{H}_t^{\ell+1}$ is nonempty, where the sets $\mathcal{H}_t^\ell$ are inductively defined as follows:

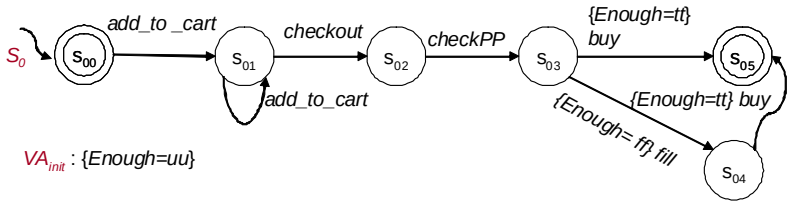
- $\mathcal{H}_t^0 = \{(s_{10}, \ldots, s_{n0}, VA_{init})\}$
- $\mathcal{H}_t^{\ell+1}$ is the set of all histories such that if $(s_1^{\ell+1}, \ldots, s_n^{\ell+1}, VA^{\ell+1})$, with $s_i^{\ell+1} = s_i'$, $h \in \mathcal{H}_t^\ell$, and $P(h, a_{\ell+1}) = i$ (with $i \neq u$), then for all transitions $(s_i^\ell, \{g\}a\{c\}, s_i') \in \delta_i$ with $g = true$ in the last variable assignment VA_h^ℓ, the history $h \cdot a_{\ell+1} \cdot (s_1^{\ell+1}, \ldots, s_n^{\ell+1}, VA^{\ell+1})$, with $s_i^{\ell+1} = s_i'$, $s_j^{\ell+1} = s_j^\ell$ for $j \neq i$, and $VA^{\ell+1} = VA_h^\ell; c$ is in $\mathcal{H}_t^{\ell+1}$.

Moreover, as before, if a trace is finite and ends after f actions, and all along all its guards are satisfied, we have that all histories in $\mathcal{H}_t^f$ end with all services in a final state. Finally, we say that an *orchestrator program P realizes the target service $\mathcal{S}_0$* if it realizes all its traces.

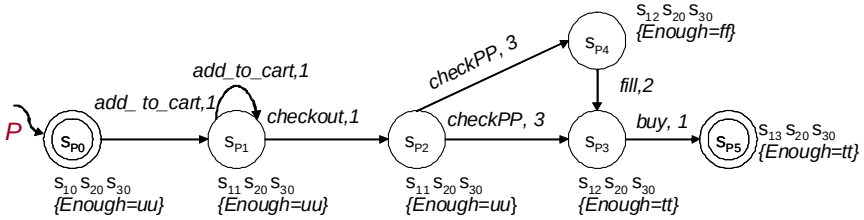
In order to understand the above definitions, let us observe that intuitively the orchestrator program realizes a trace if, as long as the guards in the trace are satisfied, it can choose at every step an available service to perform the requested action. If at a certain point a guard in the trace is not satisfied by the current variable



(a) Available Services



(b) Target Service



(c) Composition

Figure 2: Composition of nondeterministic guarded services

assignment, then we may consider the trace finished (even if not in a final state). As before, however, since when an available service executes an action it nondeterministically chooses what transition to actually perform, the orchestrator program has to require that for each of the possible resulting states of the activated available services and resulting variable assignment, the orchestrator is able to continue with the execution of the next action. Finally, the last requirement makes sure that available services are left in a final state, when a finite trace reaches its end with all guards satisfied all along.

EXAMPLE 4.3. *Referring to Examples 4.1 and 4.2, Figure 2(c) shows an orchestrator program P for the available services in Figure 2(a) that realizes the target service in Figure 2(b) from the initial assignment of $VA_{init} =$ `Enough=ff` . In the figure, for each state of P, we have represented the current state of the available services and the assignments to* `Enough`*. The orchestrator program P can be understood as follows. After delegating to $\mathcal{S}_1$ the operations* `add_to_cart` *and* `checkout`*, which do not change the value of* `Enough`*, P delegates the operation* `checkPP` *to $\mathcal{S}_3$. Since $\mathcal{S}_3$ has a nondeterministic behavior, only after its execution it is known how it changes the value of* `Enough`*. However, the orchestrator program has to be coherent with* all *possible behaviors of $\mathcal{S}_3$. Thus if* `Enough=tt` *P delegates* `buy` *to $\mathcal{S}_1$, otherwise, if* `Enough=ff` *(note that* `Enough`$\neq$`uu`*), P first delegates* `fill` *to $\mathcal{S}_2$, and since the operation* `fill` *changes the value of* `Enough` *from* `ff` *to* `tt`*, it can then finish by delegating* `buy` *to $\mathcal{S}_1$. Note that different initial assignment to* `Enough`*, may lead to different compositions.*

In order to compute an orchestrator program that realizes the target service we resort again to a reduction to satisfiability in PDL. We can use the same encoding of the previous section, now suitably modified to take into account the variable assignment. In particular, let us denote by $\mathbf{VA}$ a propositional encoding of the variable assignment VA. Then for representing the transitions of the target service $\mathcal{S}_0$ we change the formula ϕ_0 in Φ as follows:

- $s \wedge \mathbf{VA} \rightarrow \langle a \rangle \mathtt{true} \wedge [a] s'$, for each transition $(s, g, a, s') \in \delta_0$ such that g is true in VA, encoding that the target service can do an a-transition, whose guard g is satisfied, going from state s to state s'.
- $s \wedge \mathbf{VA} \rightarrow [a] undef$, for each a such that for no s' we have $(s, (g, a), s') \in \delta_0$ with g true in VA; this takes into account that the target service cannot perform an a-transition.

For representing the transitions of each available services $\mathcal{S}_i$, we change the formula ϕ_i in Φ as follows:

- $s \wedge \mathbf{VA} \wedge exec_{ia} \rightarrow \langle a \rangle (s'_1 \wedge \mathbf{VA}; \mathbf{c_1}) \wedge \cdots \wedge \langle a \rangle (s'_m \wedge \mathbf{VA}; \mathbf{c_m}) \wedge [a](s'_1 \wedge \mathbf{VA}; \mathbf{c_1} \vee \cdots \vee s'_m \wedge \mathbf{VA}; \mathbf{c_m})$, where

$\{(s'_1, c_1), \ldots, (s'_m, c_m)\} = \{(s', c) \mid (s, \{g\}a\{c\}, s') \in \delta_i$ and $g = true$ in $VA\}$, for each variable assignment VA, each $s \in S_i$, and each guarded action $\{g\}a\{c\}$ such that $(s, \{g\}a\{c\}, s') \in \delta_i$ and $g = true$ in VA; these assertions encode that if the current variable assignment is VA and a service $\mathcal{S}_i$ is in state s and is selected for the execution of an action a (i.e., $exec_{ia}$ is true), then for each possible a-transition with its guard true in VA, we have a possible a-successor in the models of Φ.

- $s \wedge \mathbf{VA} \wedge exec_{ia} \rightarrow [a]\texttt{false}$ for each variable assignment VA, and each $s \in S_i$ such that for no s', g, and c, we have that $(s, \{g\}a\{c\}, s') \in \delta_i$ with $g = true$ in VA; these denote that if the current variable assignment is VA and a service $\mathcal{S}_i$ is in state s and is selected for the execution of a but it cannot do a, then there is no a-successor in the models of Φ.

- $s \wedge \neg exec_{ia} \rightarrow [a]s$ for each state $s \in S_i$ and each action a; encoding that if service $\mathcal{S}_i$ is in state s and is not selected for the execution of a, then if a is performed (by some other available service) $\mathcal{S}_i$ does not change state.

Finally, the formula Φ is as before $Init \wedge [\mathbf{u}](\phi_0 \wedge \bigwedge_{i \in \{1,\ldots,n\}} \phi_i \wedge \phi_{add})$, where now $Init$ stands for $s_{00} \wedge s_{01} \wedge \cdots \wedge s_{0n} \wedge \mathbf{VA}_{init}$, to take into account the initial situation for the target service in the client request.

Using the PDL formula Φ defined as above, we can prove the analogs of Theorem 3.4 and Theorem 3.6.

THEOREM 4.4. *The PDL formula Φ, constructed as above, is satisfiable if and only if there exists an orchestrator program for the available services $\mathcal{S}_1, \ldots, \mathcal{S}_n$ that realizes the target service $\mathcal{S}_0$ starting from the initial variable assignment VA_{init}.*

THEOREM 4.5. *If there exists an orchestrator program for the available services $\mathcal{S}_1, \ldots, \mathcal{S}_n$ that realizes the target service $\mathcal{S}_0$ starting from the initial variable assignment VA_{init}, then there exists one that requires a finite number of states. Moreover such a finite state program can be extracted from a finite model of Φ.*

Thus we have a systematic method for synthesizing an orchestrator program for $\mathcal{S}_1, \ldots, \mathcal{S}_n$ that realizes the required target $\mathcal{S}_0$ given the initial variable assignment VA_{init}.

As for computational complexity, we observe that Φ is polynomial in the size of the target service $\mathcal{S}_0$, in the size of the available services $\mathcal{S}_i, \ldots, \mathcal{S}_n$, and in the number of possible variable assignments (and hence exponential in the number of variables in $\mathcal{V}$). So we can state the following theorem.

THEOREM 4.6. *Checking the existence of an orchestrator program for the available services $\mathcal{S}_1, \ldots, \mathcal{S}_n$ that realizes the target service $\mathcal{S}_0$ starting from the initial variable assignment VA_{init} can be done in time exponential in the size of $\mathcal{S}_1, \ldots, \mathcal{S}_n$ and $\mathcal{S}_0$ and doubly exponential in the number of variables in $\mathcal{V}$.*

5. RELATED WORK

In order to discuss *automatic* service composition, and compare different approaches, we introduce here a sort of conceptual framework for "semantic service integration", that is constituted by the following elements[15]:

- the *community ontology*, which represents the common understanding on an agreed upon reference semantics between the services [16], concerning the meaning of the offered operations, the semantics of the data flowing through the service operations, etc;

- the set of *available services*, which are the actual Web services available to the community;

- the *mapping* for the available services to the community ontology, which expresses how services expose their behavior in terms of the community ontology;

- and the *client service request*, to be expressed by using the community ontology.

To fix the idea, the setting for service composition presented in this paper can be understood in terms of this framework as follows:

- the community ontology is simple a set of actions, namely the actions of the community (and the set of shared variables in the case of guarded services);

- the available services are the actual Web services that have joined the community;

- the mapping from the available services to the community ontology is constituted by the transition systems that represent the available services; note that indeed these are expressed in terms of the actions of the community (and shared variables);

- the client service request is the target service, which again is expressed in terms of the actions of the community (and shared variables).

In general, the community ontology comprises several aspects: on one side, it describes the semantics of the information managed by the services, through appropriate semantic standards and languages (e.g., OWL and OWL-S [17], WSMO [18]); on the other side, it should consider also some specification of the service behaviors, on possible constraints and dependencies between different service operations, not limited solely to pre- and post-conditions, but considering also the process of the service.

In building such a "semantic service integration" system, two general approaches can be followed.

- In the **Service-tailored** approach, the community ontology is built mainly taking into account the available services, by suitably reconciling them; indeed the available services are directly mapped as elements of the community ontology, and the service request is composed by directly applying the mappings for accessing concrete computations.

[15]Such a framework is inspired by the research on "semantic data integration" [16, 11, 27]. Obviously that research has dealt with data (i.e., static aspects) and not with computations (i.e., dynamic aspects) that are of interest in composition of services. Still many notions and insights developed in that field may have a deep impact in service composition. An example is the distinction that we make later between "service-tailored" and "client-tailored" service integration systems, which roughly mimic the distinction between Global As View (GAV) and Local As View (LAV) in data integration.

[16]Note that many scenarios of cooperative information systems, e.g., *e*-Government or *e*-Business, consider preliminary agreements on underlying ontologies, yet yielding a high degree of dynamism and flexibility.

[17]cfr. `http://www.daml.org/services/owl-s/`.

[18]cfr. `http://www.wsmo.org/`.

- Conversely in the **Client-tailored** one, the community ontology is built mainly taking into account the client, independently from the services available; they are described (i.e., mapped) by using the community ontology, and the service request is composed by reversing these mappings for accessing concrete computations.

Again to fix the idea, it should be quite clear that the setting presented in this paper adheres to the client-tailored approach.

In fact, much of the research on automatic service composition has adopted, up to now, a service-tailored approach. For example, the works based on Classical Planning in AI (e.g., [29], [6]) consider services as atomic actions – only I/O behavior is modeled, and the community ontology is constituted by propositions/formulas (facts that are known to be true) and actions (which change the truth-value of the propositions); available services are mapped into the community ontology as atomic actions with pre- and post-conditions. In order to render a service as an atomic action, the atomic actions, as well as the propositions for pre- and post-conditions, must be carefully chosen by analyzing the available services, thus resulting in a service-tailored approach.

Other works (e.g., Papazoglou's et al. [30], Bouguettaya at al. [18], Sheth et al. [9, 8]) have essentially considered available services as atomic actions characterized by the I/O behavior and possibly effects. But differently from those based on planning, instead of concentrating on the automatic composition, they have focused more on on modeling issues and automatic discovery of services described making use of rich ontologies.

Also the work of McIlraith at al. [17] can be classified as service-tailored: services are seen as (possibly infinite) transition systems, the common ontology is a Situation Calculus Theory (therefore is semantically very rich) and service names, and each service name in the common ontology is mapped to a service seen as a procedure in Golog/Congolog Situation Calculus; the client service request is a Golog/Congolog program having service names as atomic actions with the understatement that it specifies acceptable sequences of actions for the client (as in planning) and not a transition system that the client wants to realize.

Finally, the work by Hull et al. [7, 13] describes a setting where services are expressed in terms of atomic actions (communications) that they can perform, and channels that link them with other services. The aim of the composition is to refine the behavior of each service so that the conversations realized by the overall system satisfy a given goal (dynamic property) expressed as a formula in linear time logic. Although possibly more on choreography synthesis than on composition synthesis of the form discussed here, we can still consider it a service-tailored approach, since there is no effort in hiding the service details from the client that specifies the goal formula.

Much less research has been done following a client-tailored approach, but some remarkable exceptions should be mentioned: the work of Knoblock at al. [19] is basically a data integration approach, i.e., the community ontology is the global schema of an integrated data system, the available services which are essentially data sources whose contents is mapped as views over the global schema, and the client request is basically a parameterized query over such a schema; therefore the approach is client-tailored, but neither the ontology nor mappings consider service behavior at all.

The work of Traverso et al. [26, 22, 21] can be classified also as client-tailored: services are seen as (finite) transition systems, the common ontology is a set of atomic actions and propositions, as in Planning; a service is mapped to the community ontology as a transition system using the alphabet of the community and defining how transitions affect the propositions, and the client service request asks for a sequence of actions to achieve GOAL1 (main computation), with guarantees that upon failure GOAL2 is reached (exception handling).

Finally, the line of research taken in [3, 4, 5], but also in [10], and in the present paper, has the dynamic behavior of services at the center of its investigation. In order to study the impact of such dynamics on automatic composition, all these works make simplifying assumptions on the community ontology, which essentially becomes an alphabet of actions. Still, as discussed above, the notion of community ontology is present, and in fact all these works adopt a client-tailored approach.

A fundamental issue that arises is: if such rich descriptions of the dynamic behavior of the services can be combined with rich (non propositional) descriptions of the information exchanged by the services, while keeping automatic composition feasible. The first results on this issue are reported in [2], where available services that operate on shared world description (in a form of a database) are considered. Such services can either operate on the world through some atomic processes as in OWL-S, or exchange information through messages. While the available services themselves are with finite states, the world description is not. Under suitable assumptions on how the world can be queried and modified, decidability of service composition is shown. Interestingly [2] shows that even if the available services can be modeled as deterministic transition systems, the presence of a world description whose state is not known at composition time, requires dealing with nondeterminism of the same form we have studied here.

Figure 3 summarizes, on the basis of the previous discussion, the considered works. The three axis represent the levels of detail according to which the community ontology and the mappings and the client request can be modeled. Namely, *(i) statics in the system* represents how fine grained is the modeling of the static semantics (i.e., ontologies of data and/or services, inputs and outputs, alphabet of actions, etc.); *(ii) dynamics in component services* represents how fine grained is the modeling of the processes and behavioral features of the services (only atomic actions, transition systems, etc.); and *(iii) dynamics in client service request* represents how fine grained is the modeling of the process required by the client, varying from a single step (as in the case of services consisting essentially in a single data query over a data integration system) to a (set of) sequential steps, to a (set of) conditional steps, to including loops, up to running under the full control of the client (as in our approach). *Black/white lollipops* represent service-based (white) vs. client-based (black) approaches.

6. CONCLUSION

In this paper we studied how to synthesize a composition to realize a client service request expressed as a target service, in the case where available services are only partially controllable (modeled as devilish nondeterminism) but fully observable by the orchestrator. We have shown that the problem is EXPTIME-complete and we have given effective techniques to address the problem based on PDL satisfiability. Our results extend those in [3, 4] where only deterministic available services where considered. Although not shown here, all the results in this paper can be easily extended to the case where the client request is expressed as a nondeterministic transition system as in [4]. Note that in this case the nondeterminism has a *don't-care*, aka *angelic*, nature: the client does not require to fully specify the target service he/she requires, instead he allows some degree of freedom to the composer in providing him/her with one, by choosing which one among the nondeterministic transitions to actually implement. Such a form of nondeterminism can be still tackled through a reduction to satisfiability in PDL.

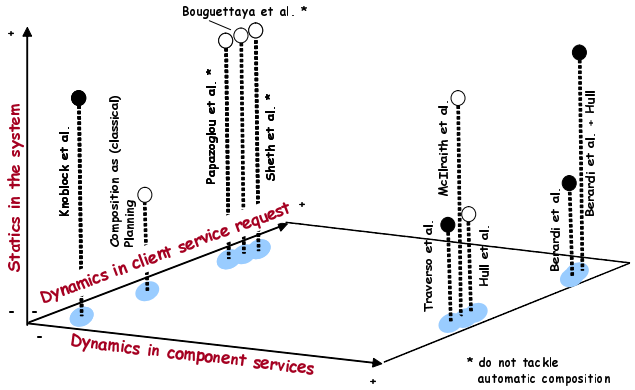


Figure 3: Comparison of the various approaches to automatic service composition

It should be noted that our approach, in which the orchestrator at each step sends an execution request to available services and these then send back to the orchestrator their states, is a form of control that is communication intensive[19]. In fact, if communication is of concern, our model is too coarse. Indeed we should distinguish between actions that affect the state of affairs and messages for sending (either contents or control) information. Suggestions on tackling such a distinction are presented in [2].

Finally, we want to stress that composition, especially in rich dynamic settings as those studied in this paper, is essentially a form of (reactive) program synthesis, and tight relationships exist with the literature on that field [23, 25, 28]. Although that literature often does not offer off-the-shelf results for composition, it certainly offers techniques and general approaches that can be profitably used to tackle subtle issues, as, for example, partial observability [15], which becomes an issue when the distinction between actions and messages is taken into account: effects of actions are not observed directly, but only communicated through messages.

7. REFERENCES

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services. Concepts, Architectures and Applications*. Springer, 2004.

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

[3] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-Services that export their behavior. In *Proc. of ICSOC 2003*.

[4] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Synthesis of underspecified composite e-Services based on automated reasoning. In *Proc. of ICSOC 2004*.

[5] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic service composition based on behavioural descriptions. *International Journal of Cooperative Information Systems*, 14(4):333–376, 2005.

[19] Actually we had essentially the same amount of control communication in [3, 4]: indeed even if states were not sent back to the orchestrator, at least some feedback for signaling the readiness to accept further commands should have been sent back.

[6] J. Blythe and J. Ambite, editors. *Proc. ICAPS 2004 Workshop on Planning and Scheduling for Web and Grid Services*, 2004.

[7] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *Proc. of WWW 2003*.

[8] J. Cardose and A. Sheth. Introduction to semantic web services and web process composition. In *Proc. 1st International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*.

[9] F. Curbera, A. Sheth, and K. Verma. Services oriented architecture and semantic web processes. In *Proc. ICWS 2004*.

[10] C. Gerede, R. Hull, O. H. Ibarra, and J. Su. Automated composition of e-services: Lookaheads. In *Proc. ICSOC 2004*.

[11] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[12] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, 2000.

[13] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-services: a look behind the curtain. In *Proc. of PODS 2003*, pages 1–14, 2003.

[14] O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *Proc. of FOCS 2005*.

[15] O. Kupferman and M. Y. Vardi. Synthesis with incomplete information. In *Proc. ICTL 1997*.

[16] M. Lenzerini. Data integration: A theoretical perspective. In *Proc. of PODS 2002*, pages 233–246, 2002.

[17] S. McIlraith and T. Son. Adapting golog for composition of semantic web services. In *Proc. KR 2002*.

[18] B. Medjahed, A. Bouguettaya, and A. Elmagarmid. Composing web services on the semantic web. *Very Large Data Base Journal*, 12(4):333351, 2003.

[19] M. Michalowski, J. Ambite, S. Thakkar, R. Tuchinda, C. Knoblock, and S. Minton. Retrieving and semantically integrating heterogeneous data from the web. *IEEE Intelligent Systems*, 19(3):72–79, 2004.

[20] A. Muscholl and I. Walukiewicz. A lower bound on web services composition. Submitted, 2005.

[21] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated composition of web services by planning at the knowledge level. In *Proc. of IJCAI 2005*, 2005.

[22] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite bpel4ws web services. In *Proc. of ICWS 2005*, 2005.

[23] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. of POPL'89*, pages 179–190, 1989.

[24] J. Rintanen. Complexity of planning with partial observability. In *Proc. of the 14th Int. Conf. on Automated Planning and Scheduling (ICAPS 2004)*, pages 345–354, 2004.

[25] W. Thomas. Infinite games and verification. In *Proc. of CAV 02*.

[26] P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *Proc. ISWC 2004*.

[27] J. D. Ullman. Information integration using logical views. *Theoretical Computer Science*, 239(2):189–210, 2000.

[28] M. Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In *Proc. of CAV 1995*.

[29] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating daml-s web services composition using shop2. In *Proc. ISWC 2003*.

[30] J. Yang and M. Papazoglou. Service components for managing the life-cycle of service compositions. *Information Systems*, 29(2):97–125, 2004.