AUTOMATIC SERVICE COMPOSITION
BASED ON BEHAVIORAL DESCRIPTIONS

DANIELA BERARDI1∗, DIEGO CALVANESE2, GIUSEPPE DE GIACOMO2,
MAURIZIO LENZERINI1 and MASSIMO MECCELLA1

1 Dipartimento di Informatica e Sistemistica “A. Ruberti”
Università di Roma “La Sapienza”
Via Salaria 113, 00198 Roma, Italy,
e-mail: lastname@dis.uniroma1.it

2 Libera Università di Bolzano/Bozen
Facoltà di Scienze e Tecnologie Informatiche
Piazza Domenici, 3, 39100 Bolzano/Bozen, Italy,
e-mail: calvanese@inf.unibz.it

Received (to be inserted)
Revised by Publisher

This paper addresses the issue of automatic service composition. We first develop a
framework in which the exported behavior of a service is described in terms of a so-
called execution tree, that is an abstraction for its possible executions. We then study
the case in which such exported behavior (i.e., the execution tree of the service) can
be represented by a finite state machine (i.e., finite state transition system). In this
specific setting, we devise sound, complete and terminating techniques both to check
for the existence of a composition, and to return a composition, if one exists. We also
analyze the computational complexity of the proposed algorithms. Finally, we present
an open source prototype tool, called E$SC$ (E-Service Composer), that implements our
composition technique. To the best of our knowledge, our work is the first attempt to
provide a provably correct technique for the automatic synthesis of service composition,
in a framework where the behavior of services is explicitly specified.

Keywords: Service, Composition, Synthesis, Behavior, Automated Reasoning

1. Introduction

Service Oriented Computing (SOC 48,2) aims at building agile networks of collabor-
ating business applications, distributed within and across organizational bound-
aries. Services (or Web Services, or e-Services, as often referred to in the literature),
which are the basic building blocks of SOC, represent a new model in the utilization
of the network: they are self-contained, modular applications that can be described,

∗(contact author)
The commonly accepted and minimal framework for services, referred to as Service Oriented Architecture (SOA), consists of the following basic roles: (i) the service provider, which is the subject (e.g., an organization) providing services; (ii) the service directory, which is the subject providing a repository/registry of service descriptions, where providers publish their services and requestors find services; and, (iii) the service requestor, also referred to as client, which is the subject looking for and invoking the service in order to fulfill some goals. A requestor discovers a suitable service in the directory, and then connects to the specific service provider in order to invoke the service.

Research on services spans over many interesting issues. In this paper, we are particularly interested in automatic service composition. Service composition addresses the situation when a client request cannot be satisfied by any available service, but a composite service, obtained by combining “parts of” available component services, might be used. The composite service can be regarded as a kind of client wrt its components, since it (indirectly) looks for and invokes them. Service composition leads to enhancements of the SOA (Extended SOA 48), by adding new elements and roles, such as brokers and integration systems, which are able to satisfy client needs by combining available services. Composition involves two different issues. The first, sometimes called composition synthesis, or simply composition, is concerned with synthesizing a new composite service, thus producing a specification of how to coordinate the component services to obtain the required service. Such a specification can be obtained either automatically, i.e., using a tool that implements a composition algorithm, or manually by a human. The second issue, often referred to as orchestration, is concerned with coordinating the various component services, and monitoring control and data flow among them, in order to guarantee the correct execution of the composite service, synthesized in the previous phase.

Our main focus in this paper is on automatic composition synthesis. In order to address this issue in an effective and well-founded way, our first contribution is a general formal framework for representing services and their behavior. Note that several works published in the literature address service oriented computing from different points of view (see the survey in Hull et al., 2003 37), but an agreed-upon comprehension of what an service is, in an abstract and general fashion, is still lacking. Often, in the literature, services are simply expressed in terms of an input/output signature, and, possibly, preconditions and effects. Our approach based on service behavioral descriptions allows the client to drive the overall execution of a service, since at each point of the computation he* can choose the next action.

*In general, the client can either be a human or another service. In what follows, we refer to the client with the “he” pronoun, in order to avoid confusion when referring to the services and to its clients using the pronouns. However, the reader should remember that we could as well as use the “it” pronoun for the client.
to perform. Note, therefore, that in our framework the focus is on actions that a service can execute; such actions can be seen as the abstractions of the effective input/output messages and operations offered by the service. In addition to a clear definition of what a service is, our framework provides a formal setting for a precise characterization of the problem of automatic composition of services.

The second contribution of the paper is an effective technique for automatic service composition. In particular, we specialize the general framework to the case where services are specified by means of finite state machines (i.e., finite state transition systems), and we present a technique that, given a specification of a target service, i.e., specified by a client, and a set of available services, synthesizes a composite service that uses only the available services, fully captures the target one, and is still described as a finite state machine. Several papers in the literature adopt finite state based formalisms as the basic models of exported behavior of services. Indeed, this class of services is particularly interesting, since they are able to carry on rather complex interactions with their clients, performing useful tasks. On the other hand, finite state formalisms represent a simple, yet powerful and widely used approach to specify the dynamic behavior of entities. We claim that most part of services have a behavior which can be abstractly represented as finite state machines. Our approach to automatic composition has two notable features:

- The composition is based on the ability of executing the available component services concurrently, and of controlling in a suitable way how such services are interleaved to serve the client.

- The client request is not a specification of a (single) desired execution, but a set of possibly non terminating executions organized in an execution tree, whose nodes correspond to sequences of transitions executed so far and whose successor nodes represent the choices available to the client to choose from what to do next. In other words, the client specifies the so-called transition system of the activities he is interested in doing. The ability of expressing a client specification as a transition systems realizes the natural client requirement that his decisions on which action to execute next depend on the outcome of previously executed actions and of other information which he cannot foresee at the time when he specifies his requests. If either the available services or the client specification are not expressed as transition systems, the client would not have any influence over the sequence of actions executed by the composite service; instead his choices would be made once and for all before the composition is performed.

Both of these features are quite distinctive of our approach, and set the stage for a quite advanced form of composition: to the best of our knowledge, here we present the first algorithm for automatic composition of services in a framework where both the available services and the client specification are characterized by a behavioral
description expressed as finite state machine. Our technique is sound, complete and terminating: if a composition of the available component services realizing the client specification exists, then our composition algorithm terminates returning one such a composition. Otherwise, it terminates reporting the non-existence of a composition. We also study the computational complexity of our technique, and we show that it runs in exponential time with respect to the size of the input state machines. While it is still an open problem assessing that such a bound is tight, we conjecture that the problem is indeed EXPTIME-hard. From a more practical point of view, it is easy to come up with examples in which the composition is exponential in the size of the component services and of the client specification, hence exponentiality is inherent to the problem*.

As third contribution of the paper, we present the prototype design and development of an open source software tool implementing our composition technique, namely ESC (e-service composer)†. Practical experimentation conducted over some real cases with the prototype shows that the tool can effectively build a composite service, despite the inherent exponential complexity of service composition, given the complexity of the behavior of real services (whose state machines are usually not too complex). We would like to remark that our automatic composition algorithm has several practical applications. In particular, in the short term, we foresee that it can constitute the core engine of semi-automatic CASE composition tools, that assist the service designer in providing the skeleton of a composite service from a set of available services. The prototype tool that we present in this paper shows exactly the feasibility and effectiveness of our algorithm.

The rest of this paper is organized as follows. In Section 2 we define the general formal framework for representing the (behavioral description of) services, the service community, i.e., the set of available services, and the problem of service composition. In Section 3 we exploit the general framework to study the case where services can be characterized by a finite number of states. In Section 4 we present a sound, complete and terminating technique for the automatic synthesis of composition. In Section 5 we present our prototype tool ESC. Finally, in Section 6 we consider related research work and in Section 7 we draw conclusions by discussing future work.

2. General Framework

A service is a software artifact (delivered over the Internet) that interacts with its clients in order to perform a specified task. A client can be either a human user, or another service. When executed, a service performs its task by directly

*Obviously, this does not give us a tight lower bound result, since the problem could be, for example, PSPACE-hard.
†cf. the PARIDE (process-based framework for composition and orchestration of dynamic e-services) Open Source Project: http://sourceforge.net/projects/paride/ that is the general framework in which we intend to release the various prototypes produced by our research.
executing certain actions, possibly interacting with other services to delegate to them the execution of other actions. In order to address SOC from an abstract and conceptual point of view, we start by identifying several facets, each one reflecting a particular aspect of a service during its life time.

- The service schema specifies the features of a service, in terms of functional and non-functional requirements. Functional requirements represent what a service does. All other characteristics of services, such as those related to quality, privacy, performance, etc. constitute the non-functional requirements. In what follows, we do not deal with non-functional requirements, and hence we use the term “service schema” to denote the specification of functional requirements only.

- The service implementation and deployment indicate how a service is realized, in terms of software applications corresponding to the service schema, deployed on specific platforms. This aspect regards the technology underlying the service implementation, and it goes beyond the scope of this paper. Therefore, although implementation issues and other related characteristics such as recovery mechanisms or exception handling, are important issues in SOC, in what follows we abstract from these properties of services.

- A service instance is an occurrence of a service effectively running and interacting with a client. In general, several running instances corresponding to the same service schema may co-exist, each one executing independently from the others.

In order to execute a service, the client needs to activate an instance of a deployed service. In our abstract model, the client can then interact with the service instance by repeatedly choosing an action and waiting for either the fulfillment of the specific task, or the return of some information. On the basis of the returned information the client chooses the next action to invoke. In turn, the activated service instance executes (the computation associated to) the invoked action; after that, it is ready to execute new actions. Under certain circumstances, i.e., when the client has reached his goal, he may explicitly end (i.e., terminate) the service instance. However, in principle, a given service instance may need to interact with a client for an unbounded, or even infinite, number of steps, thus providing the client with a continuous service. In this case, no operation for ending the service instance is ever executed. The following example gives an intuition of our approach. More details can be found in 16,13.

**Example 1.** A client wants to search and listen to mp3 files. Hence, he activates an instance of a deployed service that fulfills his needs. Once the service instance is activated and all the necessary resources for its execution are allocated, it presents the client with the set of actions that can be executed next,
namely (i) **search by author**, for searching a song by specifying its author(s), (ii) **search by title**, for searching a song by specifying its title, and (iii) **end**, for ending the interactions. The client chooses the first action and the service executes it. Again, the service presents the client with a new set of actions: let it be a singleton set, constituted by the action **listen**, for selecting and listening to a song*. Thus, the client chooses that action and the service executes it. At this point the service offers again the client with the set of actions (i), (ii), and (iii) above. The client makes his choice, for example **search by title**, and the interactions continue. When the client has reached his goal, he selects the action **end**, the service instance de-allocates all the resources associated to it and its execution ends.

Note the difference between our approach, in which we model the interactions between services and their clients through **actions**, and the approach that can be found in standard languages such as WSDL* where the focus is on exchanged **messages**. For example, in WSDL, an interaction between the service and the client is modeled by an operation, say **search by author**, with (i) a message that the client sends to the service for requesting a search, say **search by author request**, and (ii) a message that the service sends back to the client (and, in his turn, the client receives), containing the results of the computation, say **search by author response**. Hence, each WSDL operation roughly corresponds to an action in our framework.

2.1. **Service Community**

In general, when a client invokes an instance $e$, activated of a service with a schema $E$, it may happen that $e$ does not execute all of its actions on its own, but instead it **delegates** some or all of them to other (instances of) services, according to its schema. All this is transparent to the client. To precisely capture the situations when the execution of certain actions can be delegated to (instances of) other services, we introduce the notion of **community** of services:

**Definition 1. (Service Community)** A community of services is formally characterized by:

- a finite common set of actions $\Sigma$, called the **action alphabet**, or simply the **alphabet** of the community,
- a set of services specified in terms of the common set of actions.

In other words, all the services in a community **share a common understanding** over the actions in the alphabet $\Sigma$. Hence, to join a community $C$, a service needs to export its service(s) in terms of the alphabet of $C$. Also the clients interact with services in $C$ using $\Sigma$. From a more practical point of view, a community can be seen

*We assume for simplicity that the list of songs returned by **search by author** and **search by title** is non-empty.
as the set of all services whose descriptions are stored in a repository. We assume that all such service descriptions have been produced on the basis of a common and agreed upon reference alphabet/semantics. This is not a restrictive hypothesis, as many scenarios of cooperative information systems, e.g., e-Government\textsuperscript{8} or tightly-coupled e-Business\textsuperscript{25} ones, consider preliminary agreements on underlying ontologies, yet yielding a high degree of dynamism and flexibility.

The added value of a community is the fact that a service of the community may delegate the execution of some or all of its actions to other services in the community. We call such a service \textit{composite}. If this is not the case, a service is called \textit{simple}. Simple services realize offered actions directly in the software artifacts implementing them, whereas composite services, when receiving requests from clients, can (activate and) invoke other services in order to fulfill the client’s needs.

Notably, the community can be used to generate (virtual) services whose execution completely delegates actions to other members of the community. Among all the possible virtual services, in what follows we focus on the target service, i.e., (the specification of) the service the client would like to interact with, that he requests for realization to the service community. In other words, the community can be used to realize a target service requested by the client, not simply by selecting a member (i.e., a schema from which to activate an instance) of the community to which delegate the target service actions, but more generally by suitably “composing” parts of services in the community in order to obtain a virtual service which is “coherent” with the target one. This function of composing existing services on the basis of a target service is known as service composition, and is the main subject of the research reported in this paper.

\subsection*{2.2. Service Schema}

From the external point of view, i.e., that of a client, a service $E$, belonging to a community $C$, exhibits a certain \textit{exported behavior} represented as trees of atomic \textit{actions} of $C$ with constraints on their invocation order. From the internal point of view, i.e., that of an application deploying $E$ and activating and running an instance of it, it is also of interest how the actions that are part of the behavior of $E$ are effectively executed. Specifically, it is relevant to specify whether each action is executed by $E$ itself or whether its execution is delegated to another service belonging to the community $C$, transparently to the client of $E$. To capture these two points of view we introduce the notion of service schema, as constituted by two different parts, called \textit{external schema} and \textit{internal schema}, respectively.

Accordingly, service instances are characterized by an external and an internal view\textsuperscript{16}.

\subsubsection*{2.2.1. External Schema}
The aim of the external schema is to specify the exported behavior of the service. For now, in order to guarantee a general applicability of our framework, we do not refer to any particular specification formalism, rather we only assume that, whatever formalism is used, the external schema specifies the behavior in terms of a tree of actions, called external execution tree. The external execution tree abstractly represents all possible executions of a generic instance of a service. Therefore, when activated, an instance of a service executes a path of such a tree. In this sense, each node $x$ of an external execution tree represents the history of the sequence of actions of each service instance*, that has executed the path to $x$. For every action $a$ belonging to the alphabet $\Sigma$ of the community, and that can be executed at the point represented by $x$, there is a (single) successor node $x\cdot a$. The node $x\cdot a$ represents the fact that, after performing the sequence of actions leading to $x$, the client chooses to execute action $a$, among those possible, thus getting to $x\cdot a$. Therefore, each node represents a choice point at which the client makes a decision on the next action the service should perform. We call the pair $(x, x\cdot a)$ edge of the tree and we say that such an edge is labeled with action $a$. The root $\epsilon$ of the tree represents the fact that the service has not yet executed any action. Some nodes of the execution tree are final: when a node is final, and only then, the client can stop the execution of the service. In other words, the execution of a service can legally terminate only at these points†.

Notably, an execution tree does not represent the information returned to the client by the service instance execution, since the purpose of such information is to let the client choose the next action, and the rationale behind this choice depends entirely on the client.

Given the external schema $E^{ext}$ of a service $E$, we denote with $T(E^{ext})$ the external execution tree specified by $E^{ext}$.

Example 2. Figure 1 shows (a portion of) an (infinite) external execution tree characterizing the behavior of service $E_0$ (discussed in Example 1), that allows for searching and listening to mp3 files‡. In particular, the client may choose whether to search for a song by specifying either (i) its author(s) or (ii) its title (action search_by_author and search_by_title, respectively), or (iii) to terminate the service (action end, implicitly denoted by the fact that the node is final). If the client has chosen action (i) or (ii), then he selects and listens to a song (action listen). Finally, the client chooses again which action to perform next, among (i), (ii), and (iii).

2.2.2. Internal Schema

*In what follows, we omit the terms “schema” and “instance” when clear from the context.
†Typically, in a service, the root is final, to model that the computation of the service may not be started at all by the client.
‡Final nodes are represented by two concentric circles.
The internal schema specifies, besides the external behavior of the service, the information on which service instances in the community execute each given action. As before, for now we abstract from the specific formalism chosen for giving such a specification, instead we concentrate on the notion of internal execution tree. An internal execution tree is analogous to an external execution tree, except that each edge is labeled by \((a, I)\), where \(a\) is the executed action and \(I\) is a nonempty set denoting the service instances executing \(a\). Every element of \(I\) is a pair \((E', e')\), where \(E'\) is a service and \(e'\) is the identifier of an instance of \(E'\). The identifier \(e'\) unambiguously identifies the instance of \(E'\) within the service community, and, therefore, within the internal execution tree. In general, in the internal execution tree of a service \(E\), some actions may be executed also by the running instance of \(E\) itself. In this case we use the special instance identifier \texttt{this}. Note that, since \(I\) is in general not a singleton, the execution of each action can be delegated to more than one other service instance.

An internal execution tree induces an external execution tree: given an internal execution tree \(T_{\text{int}}\) we call offered external execution tree the external execution tree \(T_{\text{ext}}\) obtained from \(T_{\text{int}}\) by dropping the part of the labeling denoting the service instances, and therefore keeping only the information on the actions. An internal execution tree \(T_{\text{int}}\) conforms to an external execution tree \(T_{\text{ext}}\) if \(T_{\text{ext}}\) is equal to the offered external execution tree of \(T_{\text{int}}\).

Given a service \(E\), the internal schema \(E^\text{int}\) of \(E\) is a specification that uniquely represents an internal execution tree. We denote such an internal execution tree by \(T(E^\text{int})\).

**Definition 2. (Well-formed Service)** A service \(E\) with external schema \(E^\text{ext}\) and internal schema \(E^\text{int}\) is well-formed, if \(T(E^\text{int})\) conforms to \(T(E^\text{ext})\), i.e., its
internal execution tree conforms with its external execution tree.

We now formally define when a service of a community correctly delegates actions to other services of the community. We need a preliminary definition: given the internal execution tree $T_{int}$ of a service $E$, and a path $p$ in $T_{int}$ starting from the root, we call the projection of $p$ on an instance $e'$ of a service $E'$ the path obtained from $p$ by removing each edge whose label $(a, I)$ is such that $I$ does not contain $e'$, and collapsing start and end node of each removed edge. The notion of delegation is captured by the notion of coherency.

**Definition 3. (Coherency)** The internal execution tree $T_{int}$ of a service $E$ is coherent with a community $C$ if:

- for each edge labeled with $(a, I)$, the action $a$ is in the alphabet of $C$, and for each pair $(E', e')$ in $I$, $E'$ is a member of the community $C$;

- for each path $p$ in $T_{int}$ from the root of $T_{int}$ to a node $x$, and for each pair $(E', e')$ appearing in $p$, with $e'$ different from this, the projection of $p$ on $e'$ is a path in the external execution tree $T'_{ext}$ of $E'$ from the root of $T'_{ext}$ to a node $y$, and moreover, if $x$ is final in $T_{int}$, then $y$ is final in $T'_{ext}$.

Observe that, if a service of a community $C$ is simple, i.e., it does not delegate actions to other service instances, then it is trivially coherent with $C$. Otherwise, it is composite and hence delegates actions to other service instances. Intuitively, in the latter case, as expressed by the second bullet above, the behavior that the composite service “entails” on each component service instance must be “correct” according to the external schema of the component service instance itself.

**Definition 4. (Well-formed Community)** A community of services is well-formed if each service in the community is well-formed, and the internal execution tree of each service in the community is coherent with the community.

**Example 3.** Figure 2 shows (a portion of) an (infinite) internal execution tree*, conforming to the external execution tree of service $E_0$ shown in Figure 1, where all the actions are delegated to services of the community. In particular, the execution of search by title action and its subsequent listen action are delegated to instance $e_2$ of service $E_2$, and search by author action and its subsequent listen action to instance $e_1$ of service $E_1$.

### 2.3. Composition Synthesis

When a user requests a certain service from a service community, there may be no service in the community that can deliver it directly. However, it may still

---

*In the figure, each action is delegated to exactly one instance of a service schema. Hence, for simplicity, we have denoted a label $(a, ((E_i, e_i)))$ simply by $(a, E_i, e_i)$, for $i = 1, 2$. 
be possible to synthesize a new composite service, which suitably delegates action execution to the services of the community, and when suitably orchestrated, provides the user with the service he requested.

**Definition 5. (Composition)** Let $C$ be a well-formed service community and let $E^{\text{ext}}$ be the external schema of a target service $E$ expressed in terms of the alphabet $\Sigma$ of $C$. A *composition* of $E$ wrt $C$ is an internal schema $E^{\text{int}}$ such that:

1. $T(E^{\text{int}})$ conforms to $T(E^{\text{ext}})$,
2. $T(E^{\text{int}})$ delegates all actions to the services of $C$ (i.e., this does not appear in $T(E^{\text{int}})$), and
3. $T(E^{\text{int}})$ is coherent with $C$.

**Definition 6. (Composition Existence)** Given $C$ and $E^{\text{ext}}$, as in Definition 5, the problem of *composition existence* is the problem of checking whether there exists a composition of $E$ wrt $C$.

Observe that, since for now we are not placing any restriction of the form of $E^{\text{int}}$, the problem of composition existence corresponds to checking if there exists an internal execution tree $T_{\text{int}}$ for $E$ such that (i) $T_{\text{int}}$ conforms to $T(E^{\text{ext}})$, (ii) $T_{\text{int}}$ delegates all actions to the services of $C$, and (iii) $T_{\text{int}}$ is coherent with $C$.

**Definition 7. (Composition Synthesis)** Given $C$ and $E^{\text{ext}}$, as in Definition 5, the problem of *composition synthesis* is the problem of synthesizing an internal schema $E^{\text{int}}$ for $E$ that is a composition of $E$ wrt $C$. 

![Diagram](image-url)
3. Services with Behavioral Description as Finite State Machines

 Till now, we have not referred to any specific formalism for expressing service schemas. In what follows, we consider services whose schema (both internal and external) can be represented using only a finite number of states, i.e., using (deterministic) Finite State Machines (FSMs).

 As discussed in the introduction, several papers in the service literature adopt FSMs as the basic model of exported behavior of services \(^{20,17}\). Also, FSMs constitute the core of statecharts, which are one of the main components of UML and are becoming a widely used formalism for specifying the dynamic behavior of entities.

 In the study we report here, we make the simplifying assumption that the number of instances of a service in the community that can be involved in the internal execution tree of another service is bounded and fixed a priori. In fact, wlog we assume that it is equal to one. If more instances correspond to the same external schema, we simply duplicate the external schema for each instance. Considering that the number of services in a community is finite, this implies that the overall number of instances orchestrated in executing a service is finite and bounded by the number of services belonging to the community. Within this setting, we show how to solve the problem of composition existence, and how to synthesize a composition that is a FSM. Instead, how to deal with an unbounded number of instances remains open for future work.

 The fact that external schemas can be represented with a finite number of states means that we can factorize the sequence of actions executed up to a certain point into a finite number of states, which are sufficient to determine the future behavior of the service.

 **Definition 8. ((FSM) External Schema)** Let \( E \) be a service. The external schema of \( E \) is a FSM \( A_{E}^{ext} = (\Sigma, S_E, s_{0E}, \delta_E, F_E) \), where:

- \( \Sigma \) is the alphabet of the FSM, which is the alphabet of the community;
- \( S_E \) is the set of states of the FSM, representing the finite set of states of the service \( E \);
- \( s_{0E} \) is the initial state of the FSM, representing the initial state of the service;
- \( \delta_E : S_E \times \Sigma \to S_E \) is the (partial) transition function of the FSM, which is a partial function that given a state \( s \) and an action \( a \) returns the state resulting from executing \( a \) in \( s \);
- \( F_E \subseteq S_E \) is the set of final states of the FSM, representing the set of states that are final for the service \( E \), i.e., the states where the interactions with \( E \) can be legally terminated.

\[\blacksquare\]
Example 4. Figure 3(a) shows the external schema of the target service $E_0$ of Examples 2 and 3, specified by the client as a FSM $A_0$. Figures 3(b) and 3(c) show the external schemas, represented as FSMs $A_1$ and $A_2$, respectively associated to component services $E_1$ and $E_2$ of Example 3. In other words, $A_1$ and $A_2$ are the external schemas of the services that should be composed in order to obtain a new service that behaves like $E_0$. In particular, $E_1$ allows for searching for a song by specifying its author(s) (action \texttt{search by author}) and for listening to the song selected by the client (action \texttt{listen}). Then, it allows for executing these actions again. $E_2$ behaves like $E_1$, but it allows for retrieving a song by specifying its title (action \texttt{search by title}). $E_1$ and $E_2$ belong to the same community of services $C$. For sake of simplicity, we assume that $C$ is composed by $E_1$ and $E_2$ only, and therefore, the (finite) alphabet of actions of $C$ is $\Sigma = \{\texttt{search by author, search by title, listen}\}$. According to our setting, the client specifies the external schema $A_0$ of his target service in terms of $\Sigma$.

The FSM $A^\text{ext}_E$ is an external schema in the sense that it specifies an external execution tree $T(A^\text{ext}_E)$. Specifically, given $A^\text{ext}_E$ we define $T(A^\text{ext}_E)$ inductively on the level of nodes in the tree, by making use of an auxiliary function $\sigma(\cdot)$ that associates to each node of the tree a state in the FSM. We proceed as follows:

- $\varepsilon$, as usual, is the root of $T(A^\text{ext}_E)$ and $\sigma(\varepsilon) = s^0_E$;
- if $x$ is a node of $T(A^\text{ext}_E)$, and $\sigma(x) = s$, for some $s \in S_E$, then for each $a$ such that $s' = \delta_E(s, a)$ is defined, $x \cdot a$ is a node of $T(A^\text{ext}_E)$ and $\sigma(x \cdot a) = s'$;
- $x$ is final iff $\sigma(x) \in F_E$. 

Figure 3: Composition of services
Example 5. Figure 4 shows (a portion of the) the external execution tree $T(A_0)$ defined from $A_0$ by a mapping $\sigma$ (from nodes of $T(A_0)$ to states of $A_0$): each node of the tree is labeled with the state of $A_0$ that $\sigma$ associates to it. The mapping $\sigma$ is defined as follows.

\[
\begin{align*}
\sigma(\varepsilon) &= s_0^0 \\
\sigma(a) &= \sigma(t) = s_0^1 \\
\sigma(a \cdot l) &= \sigma(t \cdot l) = s_0^0 \\
\sigma(a \cdot l \cdot a) &= \sigma(a \cdot l \cdot t) = \sigma(t \cdot l \cdot a) = \sigma(t \cdot l \cdot t) = s_0^1 \\
\sigma(a \cdot l \cdot a \cdot l) &= \sigma(a \cdot l \cdot t \cdot l) = \sigma(t \cdot l \cdot a \cdot l) = \sigma(t \cdot l \cdot l \cdot t) = s_0^0 \\
\varepsilon &
\end{align*}
\]

$\sigma$ maps over $s_0^1$ the nodes of the tree that represent strings ending either by $a$ or by $t$; it maps over $s_0^0$ the root and the nodes of the tree associated to strings ending by $l$. Note that $T(A_0)$ coincides with the external execution tree $T_{\text{ext}}$ of Figure 1. That is, $T_{\text{ext}}$ has a finite representation as a FSM.

The external execution trees $T(A_1)$ and $T(A_2)$ for the FSMs $A_1$ and $A_2$, respectively, can be defined similarly. Finally, note that in general there may be several (equivalent) FSMs that specify the same execution tree.

Since we have assumed that each service in the community can contribute to the internal execution tree of another service with at most one instance, in specifying internal execution trees we do not need to distinguish between services and service instances. Hence, when the community $C$ is formed by $n$ services $E_1, \ldots, E_n$, it suffices to label the internal execution tree of a service $E$ by the action that caused
the transition and a subset of \([n] = \{1, \ldots, n\}\) that identifies which services in the community have contributed in executing the action. The empty set \(\emptyset\) is used to (implicitly) denote this.

We focus on internal schemas that have a finite number of states.

**Definition 9. (MFSM Internal Schema)** Given a service \(E\), we represent its internal schema as a Mealy FSM (MFSM) \(A_{E}^{\text{int}} = (\Sigma, 2^n, S_E^{\text{int}}, s_0^{\text{int}}, \delta_E^{\text{int}}, \omega_E^{\text{int}}, F_E^{\text{int}})\), where:

- \(\Sigma, S_E^{\text{int}}, s_0^{\text{int}}, \delta_E^{\text{int}}, F_E^{\text{int}}\) have the same meaning as for \(A_{E}^{\text{ext}}\);
- \(2^n\) is the output alphabet of the MFSM, which is used to denote which service(s) executes each action;
- \(\omega_E^{\text{int}} : S_E^{\text{int}} \times \Sigma \rightarrow 2^n\) is the output function of the MFSM, that, given a state \(s\) and an action \(a\), returns the subset of services that executes action \(a\) when service \(E\) is in state \(s\); if such a set is empty then \(\emptyset\) is implied; we assume that the output function \(\omega_E^{\text{int}}\) is defined exactly when \(\delta_E^{\text{int}}\) is so.

**Example 6.** Figure 5 shows a possible internal schema for the target service \(E_0\). It is represented as a MFSM \(M_0\). The output function \(\omega^{\text{int}}\) is defined as follows:

\[
\begin{align*}
\omega^{\text{int}}(s_0^0, a) &= \{1\} & \omega^{\text{int}}(s_0^0, l) &= \{1\} \\
\omega^{\text{int}}(s_0^2, t) &= \{2\} & \omega^{\text{int}}(s_0^2, l) &= \{2\}
\end{align*}
\]

The MFSM \(A_{E}^{\text{int}}\) is an internal schema in the sense that it specifies an internal execution tree \(T(A_{E}^{\text{int}})\). Given \(A_{E}^{\text{int}}\) we, again, define the internal execution tree \(T(A_{E}^{\text{int}})\) by induction on the level of the nodes, by making use of an auxiliary function \(\sigma^{\text{int}}(\cdot)\) that associates each node of the tree with a state in the MFSM, as follows:

- \(\varepsilon\) is, as usual, the root of \(T(A_{E}^{\text{int}})\) and \(\sigma^{\text{int}}(\varepsilon) = s_0^{\text{int}}\);
Example 7. Figure 6 shows a portion of the internal execution tree $T(M_0)$ defined from $M_0$, shown in Figure 5. Each node of the tree is labeled with the state of $M_0$ that $\sigma^{\text{int}}$ associates to it. The mapping $\sigma^{\text{int}}$ is defined as follows.

- $\sigma^{\text{int}}(\varepsilon) = s_0$
- $\sigma^{\text{int}}(a) = s_1$
- $\sigma^{\text{int}}(l) = s_2$
- $\sigma^{\text{int}}(a \cdot l) = \sigma^{\text{int}}(t \cdot l) = s_0$
- $\sigma^{\text{int}}(a \cdot l \cdot a) = \sigma^{\text{int}}(t \cdot l \cdot t \cdot l) = s_0$

$\sigma^{\text{int}}$ maps over $s_1$ the nodes of the tree that represent strings ending by $a$, and over $s_0$ the nodes that represent strings ending by $t$; it maps over $s_0$ the root and the nodes of the tree associated to strings ending by $l$. 

Figure 6: Internal execution tree $T(M_0)$. 

- if $x$ is a node of $T(A^{\text{int}}_E)$, and $\sigma^{\text{int}}(x) = s$, for some $s \in S^{\text{int}}_E$, then for each $a$ such that $s' = \delta^{\text{int}}_E(s, a)$ is defined, $x \cdot a$ is a node of $T(A^{\text{int}}_E)$ and $\sigma^{\text{int}}(x \cdot a) = s'$;
- if $x$ is a node of $T(A^{\text{int}}_E)$, and $\sigma^{\text{int}}(x) = s$, for some $s \in S^{\text{int}}_E$, then for each $a$ such that $\omega^{\text{int}}_E(s, a)$ is defined (i.e., $\delta^{\text{int}}_E(s, a)$ is defined), the edge $(x, x \cdot a)$ of the tree is labeled by $\omega^{\text{int}}_E(s, a)$;
- $x$ is final iff $\sigma^{\text{int}}(x) \in F^{\text{int}}_E$. 

1. if $x$ is a node of $T(A^{\text{int}}_E)$, and $\sigma^{\text{int}}(x) = s$, for some $s \in S^{\text{int}}_E$, then for each $a$ such that $s' = \delta^{\text{int}}_E(s, a)$ is defined, $x \cdot a$ is a node of $T(A^{\text{int}}_E)$ and $\sigma^{\text{int}}(x \cdot a) = s'$;
Note that $T(M_0)$ is equal to the internal execution tree $T_{\text{int}}$ of Figure 2 (up to renaming the labels $(E_i, e_i)$ with $i$). That is, $T_{\text{int}}$ has a finite representation as a MFSM. Therefore, $M_0$ is a specification of an internal execution tree that conforms to the external execution tree specified by the FSM $A_0$ of Figure 3(a). Finally, note that in general, an external FSM and its corresponding internal MFSM may have different forms.

Given a service $E$ whose external schema is an FSM and whose internal schema is an MFSM, checking whether $E$ is well formed, i.e., whether the internal execution tree conforms to the external execution tree, can be done using standard finite state machine techniques. Similarly for coherency of $E$ with a community of services whose external schemas are FSMs. In this paper, we do not go into the details of these problems, and instead we concentrate on composition.

4. Automatic Service Composition

We address the problem of checking the existence of a composite service in the FSM-based framework introduced above. We show that if a composition exists then there is one such that the internal schema is constituted by a MFSM, and we show how to actually synthesize such a MFSM, when one exists. The basic idea of our approach consists in reducing the problem of composition into satisfiability of a suitable formula of Deterministic Propositional Dynamic Logic (DPDL), a well-known logic of programs developed to verify properties of program schemas.

4.1. Deterministic Propositional Dynamic Logic

Propositional Dynamic Logics (PDLs) are a family of modal logics specifically developed for reasoning about computer programs. They capture the properties of the interaction between programs and propositions that are independent of the domain of computation. In this subsection, we provide a brief overview of a logic of this family, namely Deterministic Propositional Dynamic Logic (DPDL), which we will use in the rest of the section. More details can be found in Harel et al., 2000.

Syntactically, DPDL formulas are built by starting from a set $P$ of atomic propositions and a set $A$ of deterministic atomic actions as follows:

\[
\phi \rightarrow \text{true} \mid \text{false} \mid P \mid \neg \phi \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid \langle r \rangle \phi \mid [r] \phi
\]

\[
r \rightarrow a \mid r_1 \cup r_2 \mid r_1 ; r_2 \mid r^* \mid \phi?
\]

where $P$ is an atomic proposition in $P$, $r$ is a regular expression over the set of actions in $A$, and $a$ is an atomic action in $A$. That is, DPDL 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 latter two is, respectively, that there exists an execution of $r$ reaching a state where $\phi$ holds, and that all terminating executions of $r$ reach a state where $\phi$ holds. As far as compound programs, $r_1 \cup r_2$ means “choose non deterministically between $r_1$ and $r_2$”; $r_1 ; r_2$ means “first execute
then execute \( r_2 \); \( r^* \) means “execute \( r \) a non deterministically chosen number of times (zero or more)”; \( \phi? \) means “test \( \phi \); if it is true proceed else fail”.

The main difference between DPDL (and modal logics in general) and classical logics relies on the use of modalities. A modality is a connective which takes a formula (or a set of formulas) and produces a new formula with a new meaning. Examples of modalities are \( \langle r \rangle \) and \( [r] \). The classical logic operator \( \neg \), too, is a connective, which takes a formula \( p \) and produces a new formula \( \neg p \). The only difference is that in classical logic, the truth value of \( \neg p \) is uniquely determined by the value of \( p \), instead modalities are not truth-functional. Because of modalities, the semantics of DPDL formulas (and modal logics) is defined over a structure, namely a Kripke structure.

The semantics of a DPDL formula is based on the notion of deterministic Kripke structure. A deterministic Kripke structure is a triple of the form \( I = (\Delta^I, \{a^I\}_{a \in A}, \{P^I\}_{P \in P}) \), where \( \Delta^I \) denotes a non-empty set of states (also called worlds); \( \{a^I\}_{a \in A} \) is a family of partial functions \( a^I : \Delta^I \rightarrow \Delta^I \) from elements of \( \Delta^I \) to elements of \( \Delta^I \), each of which denotes the state transition caused by the atomic program \( a \); \( P^I \subseteq \Delta^I \) denotes all the elements of \( \Delta^I \) were \( P \) is true.

The semantic relation “a formula \( \phi \) holds at a state \( s \) of a structure \( I \)”, is written \( I, s \models \phi \), and is defined by induction on the form of \( \phi \):

\[
\begin{align*}
I, s \models \text{true} & \quad \text{always} \\
I, s \models \text{false} & \quad \text{never} \\
I, s \models P & \quad \text{iff } s \in P^I \\
I, s \models \neg \phi & \quad \text{iff } I, s \not\models \phi \\
I, s \models \phi_1 \land \phi_2 & \quad \text{iff } I, s \models \phi_1 \text{ and } I, s \models \phi_2 \\
I, s \models \phi_1 \lor \phi_2 & \quad \text{iff } I, s \models \phi_1 \text{ or } I, s \models \phi_2 \\
I, s \models \langle r \rangle \phi & \quad \text{iff } \text{there is } s' \text{ such that } (s, s') \in r^I \text{ and } I, s' \models \phi \\
I, s \models [r] \phi & \quad \text{iff } \text{for all } s', (s, s') \in r^I \text{ implies } I, s' \models \phi
\end{align*}
\]

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

\[
a^I : \Delta^I \rightarrow \Delta^I \\
(r_1 \cup r_2)^I = r_1^I \cup r_2^I \\
(r_1 ; r_2)^I = r_1^I \circ r_2^I \\
(r^*)^I = (r^I)^* \\
(\phi?)^I = \{(s, s) \in \Delta^I \times \Delta^I \mid I, s \models \phi\}
\]

It is important to understand, given a formula \( \phi \), which are the formulas that play some role in establishing the truth-value of \( \phi \). In simpler modal logics, these formulas are simply all the subformulas of \( \phi \), but due to the presence of reflexive-transitive closure (on actions) this is not the case for DPDL. Such a set of formulas is given by the Fischer-Ladner closure \( \phi? \).

A structure \( I = (\Delta^I, \{a^I\}_{a \in A}, \{P^I\}_{P \in P}) \) is called a model of a formula \( \phi \) if there exists a state \( s \in \Delta^I \) such that \( I, s \models \phi \). A formula \( \phi \) is satisfiable if there
exists a model of $\phi$, otherwise the formula is unsatisfiable. A formula $\phi$ is valid in structure $I$ if for all $s \in \Delta^I$, $I, s \models \phi$. We call axioms formulas that are used to select the interpretations of interest. Formally, a structure $I$ is a model of an axiom $\phi$, if $\phi$ is valid in $I$. A structure $I$ is a model of a finite set of axioms $\Gamma$ if $I$ is a model of all axioms in $\Gamma$. An axiom is satisfiable if it has a model and a finite set of axioms is satisfiable if it has a model. We say that a finite set $\Gamma$ of axioms logically implies a formula $\phi$, written $\Gamma \models \phi$, if $\phi$ is valid in every model of $\Gamma$. It is easy to see that satisfiability of a formula $\phi$ as well as satisfiability of a finite set of axioms $\Gamma$ can be reformulated by means of logical implication, as $\emptyset \not\models \neg \phi$ and $\Gamma \not\models \bot$ respectively.

DPDL enjoys two properties that are of particular interest (and that we will 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.

Reasoning in DPDL (and, in general, in PDLs) has been thoroughly studied from the computational point of view. In particular, the following theorem holds:

**Theorem 1.** Satisfiability in DPDL is EXPTIME-complete.

### 4.2. Checking Existence of a Composition

In this section we show how to solve the problem of composition existence.

Given the target service $E_0$ whose external schema is an FSM $A_0$ and a community of services formed by $n$ component services $E_1, \ldots, E_n$ whose external schemas are FSM $A_1, \ldots, A_n$ respectively, we build a DPDL formula $\Phi$ as follows. As set of atomic propositions $P$ in $\Phi$ we have (i) one proposition $s_j$ for each state $s_j$ of $A_j$, $j = 0, 1, \ldots, n$, denoting whether $A_j$ is in state $s_j$; (ii) propositions $F_j$, $j = 0, 1, 2$, denoting whether $A_j$ is in a final state; and (iii) propositions $\text{moved}_j$, $j = 1, 2$, denoting whether (component) FSM $A_j$ performed a transition. As set of atomic actions $A$ in $\Phi$ we have the actions in $\Sigma$ (i.e., $A = \Sigma$).

**Example 8.** As far our running example, the set $P$ of atomic propositions is defined as follows:

$$P = \{s_0^0, s_0^1, s_1^0, s_1^1, s_2^0, F_0, F_1, F_2, \text{moved}_1, \text{moved}_2\}$$

with the following meaning:

- $s_j^i$, for $i = 0, 1$ and $j = 0, 1, 2$: FSM $A_j$ is in state $s_j^i$
- $F_j$ for $j = 0, 1, 2$: FSM $A_j$ is in a final state
- $\text{moved}_j$ for $j = 1, 2$: (component) FSM $A_j$ performed a transition.
The set $A$ of deterministic atomic actions, which by construction coincides with the alphabet of the community, is defined as follows:

$$A = \Sigma = \{a, t, l\}$$

where:

- $a$ denotes action search by author
- $t$ denotes action search by title
- $l$ denotes action listen.

In order to state universal assertions, we introduce the master modality $[u]$. The formula $\Phi$ is built as a conjunction of the following formulas.

- Formulas representing $A_0 = (\Sigma, S_0, s_0, \delta_0, F_0)$:
  - $[u](s \rightarrow \neg s')$ for all pairs of states $s \in S_0$ and $s' \in S_0$, with $s \neq s'$; these say that propositions representing different states are disjoint (cannot be true simultaneously).
  - $[u](s \rightarrow \langle a \rangle true \land [a]s')$ for each $a$ such that $s' = \delta_0(s, a)$; these encode the transitions of $A_0$.
  - $[u](s \rightarrow \langle a \rangle false)$ for each $a$ such that $\delta(s, a)$ is not defined; these say when a transition is not defined.
  - $[u](F_0 \leftrightarrow \bigvee_{s \in F_0} s)$; this highlights final states of $A_0$.

**Example 9.** In our running example, we set

$$u = (a \cup t \cup l)^*$$

i.e., as the reflexive and transitive closure of the union of all atomic actions in $A$. In other words, $u$ represents the iteration of a non deterministic choice among all the possible atomic actions. Indeed, we recall that $[u]\phi$, where $\phi$ is a proposition, asserts that $\phi$ holds after any regular expression involving $a$, $t$, $l$.

Formulas capturing the external schema $A_0$ of our running example are as follows.

$$[u](s_0^0 \rightarrow \neg s_0^1)$$

This formula states that FSM $A_0$ can never be simultaneously in the two states $s_0^0$ and $s_0^1$. Note that it is equivalent to state $[u](s_0^1 \rightarrow \neg s_0^0)$.

$$[u](s_0^0 \rightarrow \langle a \rangle true \land [a]s_0^1)$$

$$[u](s_0^0 \rightarrow \langle t \rangle true \land [t]s_0^1)$$

$$[u](s_0^0 \rightarrow \langle l \rangle true \land [l]s_0^1)$$
These formulas encode the transitions that\( A_0 \) can perform. For example, the first formula asserts that, for all possible sequence of actions, if \( A_0 \) is in state \( s_0^0 \), the FSM allows for searching an mp3 file by author, i.e., it can execute action \( a \), and it necessarily moves to state \( s_1^0 \). Analogously for the other formulas:

\[
\begin{align*}
[u](s_0^0 & \rightarrow [\ell]false) \\
[u](s_1^0 & \rightarrow [a]false) \\
[u](s_0^0 & \rightarrow [\ell]false)
\end{align*}
\]

These formulas encode the transitions that are not defined on \( A_0 \). For example, the first formula asserts that, for all possible sequences of actions, it is never possible to execute action \( \text{listen} \) when the FSM is in state \( s_0^0 \).

\[
[u](F_0 \leftrightarrow s_0^0)
\]

Finally, this formula asserts that \( s_0^0 \) is a final state for \( A_0 \).

- Formulas encoding each component FSM \( A_i = (\Sigma, S_i, s_i^0, \delta_i, F_i) \):
  - \([u](s \rightarrow \neg s')\) for all pairs of states \( s \in S_i \) and \( s' \in S_i \), with \( s \neq s' \); these again say that propositions representing different states are disjoint.
  - \([u](s \rightarrow [a](\text{moved}_i \land s' \lor \neg\text{moved}_i \land s))\) for each \( a \) such that \( s' = \delta_i(s, a) \); these encode the transitions of \( A_i \), conditioned to the fact that the component \( A_i \) is actually required to make a transition \( a \) in the composition.
  - \([u](s \rightarrow [a](\neg\text{moved}_i \land s))\) for each \( a \) such that \( \delta_i(s, a) \) is not defined; these say that when a transition is not defined, \( A_i \) cannot be asked to execute it in the composition, and therefore \( A_i \) does not change state.
  - \([u](F_i \leftrightarrow \bigvee_{s \in F_i} s)\); this highlights final states of \( A_i \).

**Example 10.** Formulas capturing the external schema \( A_1 \) of our running example.

\[
[u](s_0^1 \rightarrow \neg s_1^1)
\]

This formula has an analogous meaning as that relative to \( A_0 \).

\[
\begin{align*}
[u](s_0^1 & \rightarrow [a](\text{moved}_1 \land s_1^1 \lor \neg\text{moved}_1 \land s_0^0)) \\
[u](s_1^1 & \rightarrow [\ell](\text{moved}_1 \land s_0^0 \lor \neg\text{moved}_1 \land s_1^1))
\end{align*}
\]

These formulas encode the transitions of \( A_1 \), conditioned to the fact that component \( A_1 \) is actually required to make a transition in the composition. As an example, the first formula asserts that for all possible sequences of actions, if the FSM \( A_1 \) is in \( s_1^0 \), then after action \( a \) has been executed, necessarily one of the following conditions must hold: either it is \( A_1 \) that performed the transition and therefore it moved to state \( s_1^1 \), or the transition has been
performed by another FSM, hence $A_1$ did not move and remained in the current state $s^0_1$. 

$$ [u](s^0_1 \rightarrow [l](\neg\text{moved}_1 \land s^0_1)) $$
$$ [u](s^0_1 \rightarrow [l](\neg\text{moved}_1 \land s^0_1)) $$
$$ [u](s^1_1 \rightarrow [a](\neg\text{moved}_1 \land s^1_1)) $$
$$ [u](s^1_1 \rightarrow [l](\neg\text{moved}_1 \land s^1_1)) $$

These formulas encode the situation when a transition is not defined. For example, the first formula states that if the FSM is in state $s^0_1$ and it receives actions $l$ in input, it does not move, and therefore it remains in state $s^0_1$; this holds for all possible (previous) sequences of actions. Note that the situation when the FSM does not move is different from the situation when it loops on a state: indeed, in the latter case the transition is defined whereas in the former it does not.

Finally, the formula

$$ [u](F_1 \leftrightarrow s^0_1) $$

asserts that state $s^0_1$ is final for FSM $A_1$.

Formulas capturing the external schema $A_2$ of our running example.

Such formulas are analogous to the previous ones, therefore, we will just report them, without further comments.

$$ [u](s^0_2 \rightarrow \neg s^1_2) $$
$$ [u](s^0_2 \rightarrow [l](\text{moved}_2 \land s^1_2 \lor \neg\text{moved}_2 \land s^0_2)) $$
$$ [u](s^0_2 \rightarrow [l](\text{moved}_2 \land s^1_2 \lor \neg\text{moved}_2 \land s^0_2)) $$
$$ [u](s^0_2 \rightarrow [l](\neg\text{moved}_2 \land s^1_2)) $$
$$ [u](s^1_2 \rightarrow [a](\neg\text{moved}_2 \land s^0_2)) $$
$$ [u](s^1_2 \rightarrow [l](\neg\text{moved}_2 \land s^1_2)) $$
$$ [u](s^1_2 \rightarrow [a](\neg\text{moved}_2 \land s^1_2)) $$
$$ [u](F_2 \leftrightarrow s^0_2) $$

- Finally, formulas encoding domain independent conditions:

  $- s^0_0 \land \bigwedge_{i=1,...,n} s^0_i$; this says that initially all services are in their initial state; note that this formula is not prefixed by $[u](\cdot)$.

  $- [u]([a]\text{true} \rightarrow [a] \bigvee_{i=1,...,n} \text{moved}_i)$, for each $a \in \Sigma$; these say that at each step at least one of the component FSM has moved.

  $- [u](F_0 \rightarrow \bigwedge_{i=1,...,n} F_i)$; this says that when the target service is in a final state also all component services must be in a final state.
Example 11. The following formulas must hold for the overall composition of our running example.

\[ s^0_0 \land s^0_1 \land s^0_2 \]

It asserts that all services start from their initial states.

\[
\begin{align*}
[u](\langle a \rangle \text{true} & \rightarrow [a](\text{moved}_1 \lor \text{moved}_2)) \\
[u](\langle t \rangle \text{true} & \rightarrow [t](\text{moved}_1 \lor \text{moved}_2)) \\
[u](\langle l \rangle \text{true} & \rightarrow [l](\text{moved}_1 \lor \text{moved}_2))
\end{align*}
\]

Each formula expresses that at each step at least one FSM moves. For example, the first one asserts that for all possible execution sequences, if execution of \(a\) terminates, then necessarily \(a\) is executed by at least one component service, either \(E_1\) or \(E_2\).

Finally,

\[ [u](F_0 \rightarrow F_1 \land F_2) \]

states that if the composite service is in a final state, both component services must be in a final state: the composite service may legally terminate only if also all the component services can.

Lemma 1. If there exists a composition of \(E_0\) wrt \(E_1, \ldots, E_n\), then the DPDL formula \(\Phi\), constructed as above, is satisfiable.

Proof. Suppose that there exists some internal schema (without restriction on its form) \(E_0^{\text{int}}\) which is a composition of \(E_0\) wrt \(E_1, \ldots, E_n\). Let \(T_{\text{int}} = T(E_0^{\text{int}})\) be the internal execution tree defined by \(E_0^{\text{int}}\).

Then for the target service \(E_0\) and each component service \(E_i, i = 1, \ldots, n\), we can define mappings \(\sigma\) and \(\sigma_i\) from nodes in \(T_{\text{int}}\) to states of \(A_0\) and \(A_i\), respectively, by induction on the level of the nodes in \(T_{\text{int}}\) as follows.

- base case: \(\sigma(\varepsilon) = s^0_0\) and \(\sigma_i(\varepsilon) = s^0_i\).
- inductive case: let \(\sigma(x) = s\) and \(\sigma_i(x) = s_i\), and let the node \(x \cdot a\) be in \(T_{\text{int}}\) with the edge \((x, x \cdot a)\) labeled by \((a, I)\), where \(I \subseteq [n]\) and \(I \neq \emptyset\) (notice that this may not occur since \(T_{\text{int}}\) is specified by a composition). Then we define

\[
\sigma(x \cdot a) = s' = \delta_0(s, a)
\]

and

\[
\sigma_i(x \cdot a) = \begin{cases} 
  s_i' = \delta_i(s, a) & \text{if } i \in I \\
  s_i & \text{if } i \notin I
\end{cases}
\]

Once we have \(\sigma\) and \(\sigma_i\) in place we can define an interpretation \(\mathcal{I} = (\Delta^\mathcal{I}, \{a^\mathcal{I}\}_{a \in \Sigma}, \{P^\mathcal{I}\}_{P \in \mathcal{P}})\) for \(\Phi\) as follows:

- \(\Delta^\mathcal{I} = \{x \mid x \in T_{\text{int}}\};\)
\begin{itemize}
  \item $a^I = \{(x, x \cdot a) \mid x, x \cdot a \in T_{int}\}$, for each $a \in \Sigma$;
  \item $s^I = \{x \in T_{int} \mid \sigma(x) = s\}$, for all propositions $s$ corresponding to states of $A_0$;
  \item $s_i^I = \{x \in T_{int} \mid \sigma_i(x) = s_i\}$, for all propositions $s_i$ corresponding to states of $A_i$;
  \item $moved^I_i = \{x \cdot a \mid (x, x \cdot a) \text{ is labeled by } I\}$, for $i = 1, \ldots, n$;
  \item $F_0^I = \{x \in T_{int} \mid \sigma(x) = s \text{ with } s \in F_0\}$;
  \item $F_i^I = \{x \in T_{int} \mid \sigma_i(x) = s_i \text{ with } s_i \in F_i\}$, for $i = 1, \ldots, n$.
\end{itemize}

Since $T_{int}$ is a composition of $E_0$ wrt $E_1, \ldots, E_n$, it is easy to check that the interpretation $I$ built as above, is a model for $\Phi$ and that, therefore, $\Phi$ is satisfiable. 

**Lemma 2.** Any model of the DPDL formula $\Phi$, constructed as above, denotes a composition of $E_0$ wrt $E_1, \ldots, E_n$.

**Proof.** Suppose $\Phi$ is satisfiable. For the tree model property, there exists a tree-like model for $\Phi$: let $I = (\Delta^I, \{a^I\}_{a \in \Sigma}, \{P^I\}_{P \in \mathcal{P}})$ be such a model. From $I$ we can build an internal execution tree $T_{int}$ for $E_0$ as follows.

\begin{itemize}
  \item the nodes of the tree are the elements of $\Delta^I$; actually, since $I$ is tree-like we can denote the elements in $\Delta^I$ as nodes of a tree, using the same notation that we used for internal/external execution tree;
  \item nodes $x$ such that $x \in F_0^I$ are the final nodes;
  \item if $(x, x \cdot a) \in a^I$ and for all $i \in I$, $x \cdot a \in moved^I_i$ and for all $j \notin I$, $x \cdot a \notin moved^I_j$, then $(x, x \cdot a)$ is labeled by $(a, I)$.
\end{itemize}

It is straightforward to show that: (i) $T_{int}$ conforms to $T(A_0)$, (ii) $T_{int}$ delegates all actions to the services of $E_1, \ldots, E_n$, and (iii) $T_{int}$ is coherent with $E_1, \ldots, E_n$.

Since we are not placing any restriction on the kind of specification allowed for internal schemas, it follows that there exists an internal schema $E_{int}$ that is a composition of $E_0$ wrt $E_1, \ldots, E_n$. 

**Theorem 2.** The DPDL formula $\Phi$, constructed as above, is satisfiable if and only if there exists a composition of $E_0$ wrt $E_1, \ldots, E_n$.

**Proof.** Straightforward, from Lemma 1 and 2.

Observe that the size of $\Phi$ is polynomially related to $A_0$ and $A_1, \ldots, A_n$. Hence, from the EXPTIME-completeness of satisfiability in DPDL and from Theorem 2 we get the following complexity result.

**Theorem 3.** Checking the existence of a service composition can be done in EXPTIME.
4.3. **Synthesizing a Composition**

In the previous section we have shown that we are able to check the existence of a composition by checking satisfiability of a DPDL formula $\Phi$ encoding the target service, the services in the community and a number of domain independent conditions. In this section we extend our technique to actually synthesize a composition which is an FSM. Specifically, we present an algorithm that returns a composition, if one exists, and returns a special symbol (nil), denoting that no composition exists, otherwise.

Intuitively, by Theorem 2, if $\Phi$ is satisfiable then it admits a model, which is exactly the internal schema, i.e., the composition we want to synthesize. Conversely, if $\Phi$ is not satisfiable, no model exists, therefore, the component FSM $A_1, \ldots, A_n$ cannot be composed in order to achieve the target FSM $A_0$. Note that Theorem 2 says nothing about compositions which are finite state machines. However, because of the small model property, from the DPDL formula $\Phi$ one can always obtain a model which is at most exponential in the size of $\Phi$. From such a model one can extract an internal schema for $E_0$ that is a composition of $E_0$ wrt $E_1, \ldots, E_n$, and which has the form of a MFSM.

**Definition 10. (Mealy Composition)** Given a finite model $I_f = (\Delta_{I_f}, \{a_{I_f}^i\}_{a \in \Sigma}, \{P_{I_f}^i\}_{P \in P})$, we define *Mealy composition* an MFSM $A_c = (\Sigma, 2^{[n]}, S_c, s_0^c, \delta_c, \omega_c, F_c)$, built as follows:

- $S_c = \Delta_{I_f}$;
- $s_0^c = d_0$ where $d_0 \in (s_0^0 \land \bigwedge_{i=1,\ldots,n} s_i^0)^{I_f}$;
- $s' = \delta_c(s, a)$ iff $(s, s') \in a_{I_f}^{T_f}$;
- $I = \omega_c(s, a)$ iff $(s, s') \in a_{I_f}^{T_f}$ and for all $i \in I$, $s' \in moved_i^{T_f}$ and for all $j \notin I$, $s' \notin moved_j^{T_f}$;
- $F_c = F_0^{T_f}$.

As a consequence of this, we get the following results.

**Theorem 4.** If there exists a composition of $E_0$ wrt $E_1, \ldots, E_n$, then there exists a Mealy composition whose size is at most exponential in the size of the external schemas $A_0, A_1, \ldots, A_n$ of $E_0, E_1, \ldots, E_n$ respectively.

**Proof.** By Theorem 2, if $A_0$ can be obtained by composing $A_1, \ldots, A_n$, then the DPDL formula $\Phi$ constructed as above is satisfiable. In turn, if $\Phi$ is satisfiable, for the small-model property of DPDL there exists a model $I_f$ of size at most exponential in $\Phi$, and hence in $A_0$ and $A_1, \ldots, A_n$. From $I_f$ we can construct a MFSM $A_c$ as above. The internal execution tree $T(A_c)$ defined by $A_c$ satisfies all the conditions required for $A_c$ to be a composition, namely: (i) $T(A_c)$ conforms to...
AUTOMATIC SERVICE COMPOSITION

1 INPUT: $A_0$ /* FSM external schema of target service */

2 $A_1 \ldots A_n$ /* FSM external schema of services in the community */

3 OUTPUT: if (a composition of $A_0$ wrt $A_1 \ldots A_n$ exists)

4 then return a Mealy composition of $A_0$ wrt $A_1 \ldots A_n$

5 else return nil

6 begin

7 $Φ :=$ FSM2DPDL$(A_0, A_1, \ldots A_n)$;

8 $I_f :=$ DPDLTableau$(Φ)$;

9 if ($I_f ==$ nil)

10 then return nil

11 else $A_c :=$ ExtractMFSM$(I_f)$;

12 $C_{min} :=$ Minimize$(A_c)$;

13 return $C_{min}$;

14 end

Figure 7: The Algorithm for Synthesizing Mealy Composition

$T(A_0)$, (ii) $T(A_c)$ delegates all actions to the services of $E_1, \ldots, E_n$, and (iii) $T(A_c)$ is coherent with $E_1, \ldots, E_n$.

Theorem 5. Any finite model of the DPDL formula $Φ$ denotes a Mealy composition of $E_0$ wrt $E_1, \ldots, E_n$.

Proof. By construction, observing that the construction of the Mealy composition from a finite model is semantic-preserving.

Figure 7 shows our algorithm, which consists of the following steps. First (line 9), the DPDL formula $Φ$ is built, exploiting the FSM2DPDL function, as a conjunction of formulas encoding: (i) the target service requested by the client, (ii) the (available) services of the community, and (iii) domain independent conditions. In other words, it encodes all (real and virtual) services participating in the composition. Essentially, such an encoding aims at characterizing which service in the community “moves” in correspondence with each transition of the target service, so that general domain independent conditions are satisfied. The novelty and peculiarity of our approach to service composition is exactly this: we delegate to one or more services in the community the execution of each action present in the client specification, since only in a second moment it is known which actions will be chosen by the client for execution (and the composite service should be able to execute any action chosen by the client). Satisfiability of $Φ$ is then checked (line 10, function DPDLTableau) exploiting tableau algorithms that return a (finite) model, if one exists. If $Φ$ is not satisfiable, no model exists, and our algorithm returns nil (line 12). Otherwise, from a finite model a Mealy composition is built, (function ExtractMFSM, line 13), according to Definition 10. Intuitively, the transformation from a finite model $I_f$ to a Mealy Machine $A_c$ consists in discarding from each state of $I_f$ the
information about the current state of each component service, therefore keeping in $A_c$ only the information about which service is in a final state and which one “moves”. Note that, in general, after this transformation, some states of $A_c$ can be redundant, since they contain the same information: in other words, a final step minimizing $A_c$ can be performed (line 14, function $\text{Minimize}$), and the minimal Mealy composition $C_{\min}$ is returned (line 15). As we will show in Section 5, our prototype tool implements exactly such steps.

**Example 12.** Let $\Phi$ be the DPDL formula encoding $A_0$, $A_1$, $A_2$ and the domain independent conditions, built as in Section 4.2. Let $I_f$ be the finite model (i.e., the Kripke Stricture) obtained using a tableau technique for DPDL. $I_f$ is defined as $I_f = (\Delta^f, \{a^f\}_{a \in \Sigma}, \{P^f\}_{P \in \mathcal{P}})$, where:

$$\begin{align*}
\Delta^f &= \{(t_0, t_1, t_2, t_3, t_4) \} \\
\{a^f\}^f &= \{((t_0, t_1), (t_2, t_1), (t_4, t_1))\} \\
\{P^f\}^f &= \{((t_0, t_2), (t_3, t_4))\} \\
\{s^f_0\}^f &= \{(t_0, t_2, t_4)\} \\
\{s^f_1\}^f &= \{(t_1, t_3)\} \\
\{s^f_2\}^f &= \{(t_0, t_2, t_3, t_4)\} \\
\{s^f_3\}^f &= \{(t_1)\}
\end{align*}$$

Each state $t_i$ of the model is associated with the atomic propositions in $\mathcal{P}$ that hold in that state, according to $I_f$. For example, consider state $t_0$ (which is initial for the model): $I_f$ imposes that $s^0_f \land s^1_f \land s^2_f \land F_0 \land F_1 \land F_2$ holds in $t_0$. For sake of readability, in the figure we have associated to each state of $I_f$ simply the list of atomic propositions that are true. Additionally, note that the DPDL encoding does not pose any constraint on the value of $\text{moved}_i$ predicates in the initial state of the model: their value has been arbitrarily chosen to be $\text{false}$.*

*Note also the model for the DPDL formula $\Phi$ is deterministic, as it should be. Non determinism could have been introduced by the operator $. However, we are guaranteed that no atomic
Given $I_f = (\Delta_{I_f}, \{a^{I_f}\}_a \in \Sigma_1, \{P^{I_f}\}_P \in \mathcal{P})$ of $\Phi$, we define a Mealy Machine $A_c = (\Sigma, 2^{[n]}, S_c, s^0_c, \delta_c, \omega_c, F_c)$ representing the internal schema of the target service, as follows:

- $S_c = \{t_0, t_1, t_2, t_3, t_4\}$;
- $s^0_c = t_0$, where $t_0 \in (s^0_0 \land s^0_1 \land s^0_2)^{I_f}$; note that we could have as well as chosen either $t_2$ or $t_4$ as initial state.
- $\delta_c$ is defined as:
  \[
  \begin{align*}
  \delta_c(t_0, a) &= t_1 \\
  \delta_c(t_0, t) &= t_3 \\
  \delta_c(t_1, l) &= t_2 \\
  \delta_c(t_3, l) &= t_4 \\
  \delta_c(t_2, a) &= t_1 \\
  \delta_c(t_2, t) &= t_3 \\
  \delta_c(t_4, a) &= t_1 \\
  \delta_c(t_4, t) &= t_3
  \end{align*}
  \]
- $\omega_c$ is defined as:
  \[
  \begin{align*}
  \omega_c(t_0, a) &= \{1\} \\
  \omega_c(t_0, t) &= \{2\} \\
  \omega_c(t_1, l) &= \{1\} \\
  \omega_c(t_3, l) &= \{2\} \\
  \omega_c(t_2, a) &= \{1\} \\
  \omega_c(t_2, t) &= \{2\} \\
  \omega_c(t_4, a) &= \{1\} \\
  \omega_c(t_4, t) &= \{2\}
  \end{align*}
  \]
- $F_c = \{t_0, t_2, t_4\}$.

This example shows also that the finite state machine associated to the finite model of $\Phi$ is in general not minimal. Indeed, the minimal MFSM $C_{min}$ is shown in Figure 9. Note that $C_{min}$ coincides with the MFSM shown in Figure 5 which, as shown in Example 7, is an internal schema for the target service $E_0$ of our running example.

\[
\begin{align*}
a &= \text{search by author} \\
t &= \text{search by title} \\
l &= \text{listen}
\end{align*}
\]
Finally, note that our composition algorithm can be easily extended to produce compositions satisfying additional constraints expressed in DPDL, for instance, we may require that once a certain state of the composite service is reached, it is never reached again. The algorithm in Figure 7 can be extended as follows. It takes in input also a DPDL formula $\Phi_{prop}$ encoding the additional constraints that the composition should satisfy. Line 10 is replaced with $I_f := \text{DPDLTableau}(\Phi \land \Phi_{prop})$, satisfiability of the conjunct $\Phi \land \Phi_{prop}$ is checked, and a model $I_f$ is returned if one exists. It is easy to see that any model $I_f$ is a composition of the available services, that realizes the target services and that satisfies the required constraints. The inclusion of additional constraints in our encoding goes beyond the scope of this paper and will not be further addressed.

5. The Service Composition Tool $\mathcal{ESC}$

In this section we discuss the prototype tool $\mathcal{ESC}$ that we developed to compute automatic service composition in our framework.

Figure 10 shows the high level architecture for $\mathcal{ESC}$. Each service is represented in terms of both its static interface, through a WSDL document, and its behavioral description*, which can be expressed in any language that allows to express a finite state machine (e.g., Web Service Conversation Language 32, Web Service Transition Language 17, BPEL4WS 3, etc.). We recall that in our framework the focus is on actions that a service can execute; such actions can be seen as the abstractions of the effective input/output messages and operations offered by the service. As an example, Figure 11 shows the WSDL interface of service $E_0$ whose behavior is represented in Figure 3(a).

*Note that such behavioral description of services specifies the external schema.
<definitions xmlns:y="http://new.thiswebservice.namespace"
targetNamespace="http://new.thiswebservice.namespace">

<!-- Types -->
<types>
  <element name="ListOfSong_Type">
    <complexType>
      <sequence>
        <element minOccurs="1"
          maxOccurs="unbound"
          name="SongTitle"
          type="xs:string"/>
      </sequence>
    </complexType>
  </element>
</types>

<!-- Messages -->
<message name="search_by_title_request">
  <part name="containedInTitle" type="xs:string"/>
</message>
<message name="search_by_title_response">
  <part name="matchingSongs" xsi:type="ListOfSong_Type"/>
</message>
<message name="search_by_author_request">
  <part name="authorName" type="xs:string"/>
</message>
<message name="search_by_author_response">
  <part name="matchingSongs" xsi:type="ListOfSong_Type"/>
</message>
<message name="listen_request">
  <part name="selectedSong" type="xs:string"/>
</message>
<message name="listen_response">
  <part name="MP3fileURL" type="xs:string"/>
</message>

<!-- Service and Operations -->
<portType name="MP3CompositeServiceType">
  <operation name="search_by_title">
    <input message="y:search_by_title_request"/>
    <output message="y:search_by_title_response"/>
  </operation>
  <operation name="search_by_author">
    <input message="y:search_by_author_request"/>
    <output message="y:search_by_author_response"/>
  </operation>
  <operation name="listen">
    <input message="y:listen_request"/>
    <output message="y:listen_response"/>
  </operation>
</portType>
</definitions>

Figure 11: WSDL specification of service $E_0$ whose external schema $A_0$ is represented in Figure 3(a).
We start from a repository of services, which implements the community of services, and which can be seen, therefore, as an advanced version of UDDI\textsuperscript{55}. The client specifies his target service in terms of a WSDL document and of its behavioral description, again expressed using one of the languages mentioned before\textsuperscript{∗}. Both the services in the repository and the target service are then abstracted into the corresponding FSM (Abstraction Module). The Synthesis Engine is the core module of \(\mathcal{ESC}\). It takes in input such FSMs, produces the DPDL formula \(\Phi\), (possibly) builds a model and produces in output the MFSM of the composite service, where each action is annotated with (the identifier of) the component service(s) that executes it. Finally, such abstract version of the composite service is realized into a BPEL4WS specification\textsuperscript{†} (Realization Module), that can be executed by an orchestration engine, i.e., a software module that suitably coordinates the execution of the component services participating to the composition\textsuperscript{2}.

We tested our tool on several examples, involving communities containing up to 10 services, each one having roughly 10-20 states: \(\mathcal{ESC}\) performs quite nicely, considering that the current release does not implement any relevant optimization.

The implementation of the Abstraction Module depends on which language is used to represent the behavioral description of services. In our prototype we use Web Service Transition Language, which is translated into FSMs\textsuperscript{17}.

In the next subsections we will provide some details on the Synthesis Engine and the Realization Module.

5.1. Implementation of the Synthesis Engine Module

From a practical point of view, in order to actually synthesize a Mealy composition, we resort to Description Logics (DLs\textsuperscript{6}), exploiting the well known correspondence between DPDL formulas and DL knowledge bases\textsuperscript{4}. Tableaux algorithms for DLs have been widely studied in the literature, therefore, one can use current highly optimized DL-based systems\textsuperscript{36,33} to check the existence of service compositions. However, such systems cannot be used to synthesize a Mealy composition because they do not return a model. Therefore, we implemented from scratch a tableau algorithm for DL that builds a model\textsuperscript{15} (of the DL knowledge base that encodes the specific composition problem) which is a Mealy composition. For our purpose the well-known \(\mathcal{ALC}\)\textsuperscript{6}, equipped with the ability of expressing axioms, suffices\textsuperscript{14}.

The various functionalities of the Synthesis Engine are implemented into three Java sub-modules.

- The FSM2ALC Translator module takes in input the FSMs produced by the

\textsuperscript{∗}The behavioral description of both the client specification and the services in the repository are expressed in the same language.

\textsuperscript{†}It represents the internal schema for the target service.

\textsuperscript{†}In fact, current Description Logics systems cannot handle Kleene star. However, since in our DPDL formula \(\Phi\), \(\ast\) is only used to mimic universal assertions, and such systems have the ability of handling universal assertions, they can indeed check satisfiability of \(\Phi\).

\textsuperscript{§}If one exists.
**Abstraction Module**, and translates them into an $\mathcal{ALC}$ knowledge base (details of the encoding are presented in 14).

- The $\mathcal{ALC}$ Tableau Algorithm module implements the standard tableau algorithm for $\mathcal{ALC}$ (cf., Buchheit et al., 1993 19). It takes in input the $\mathcal{ALC}$ knowledge base and checks its satisfiability, or, equivalently, it verifies if a composition exists. If this is the case, it returns a model of the knowledge base, which is a finite state machine. Otherwise, it returns the information about unsatisfiability of the knowledge base, i.e., the non-existence of a composition.

- The FSM Minimizer module minimizes the model, since it may contain states which are unreachable or unnecessary. Classical, standard minimization techniques can be used, in particular, we implemented the Implication Chart Method 51. The minimized FSM is then converted into a Mealy FSM, where each action is annotated with the service in the repository that executes it.

Since these three modules are in fact independent, they are wrapped into an additional module, the Composer Module, which also provides the external interface.

### 5.2. Implementation of the Realization Module

The Realization Module, whose development is currently ongoing, is in charge of producing an executable BPEL4WS file starting from the automatically synthesized MFSM. In the following, we outline the intuitions that are driving our design and development (based on results in 7,15):

- Transitions are mapped first, thus deriving transition skeletons, then states are mapped, thus deriving state skeletons, and finally the BPEL4WS file is obtained, by connecting state skeletons on the basis of the MFSM; in such a way the obtained BPEL4WS specification has a structure similar to the one shown in Figure 12, i.e., with a <flow> operation wrapping all the state skeletons, connected among them by <link>s.

- Each transition corresponds to a BPEL4WS pattern (i.e., transition skeleton) consisting of (i) an <onMessage> operation (in order to wait for the input from the client of the composite service), (ii) followed by the invocation to the appropriate component service, and then (iii) a final operation for returning the result to the client. Of course both before the component service invocation and before returning the result, messages should be copied forth and back in appropriate variables.

- All the transitions originating from the same state are collected in a <pick> operation, having as many <onMessage> clauses as transitions originating from the state; this is the state skeleton.
• The above steps for transition and state skeletons work for request/reply interactions; simple modifications are needed for notification/response, one-way and notification-only interactions, that can imply a proactive behaviour of the composite service, possibly guarded by \texttt{<onAlarm>} blocks. Figure 13 shows the structure of the skeletons.

• Finally, the BPEL4WS file is built visiting the MFSM in depth, starting from the initial state and applying the previous rules. Specifically, all the \texttt{<pick>} blocks are enclosed in a surrounding \texttt{<flow>}; the dependencies are modeled as \texttt{<link>}s: \texttt{<link>}s are controlled by specific variables $S_i$-to-$S_j$ that are set to \texttt{TRUE} iff the transition $S_i \rightarrow S_j$ is executed; each state skeleton has many outgoing \texttt{<link>}s as states connected in output, each going to the appropriate \texttt{<pick>} block.

• The previous step works for acyclic state machines. In the case of a state machine with cycles, the following intuition can be applied: (i) identify all the cycles; (ii) for each cycle enclose the involved state skeletons inside a \texttt{<while>} block controlled by a condition $\neg$\texttt{exit}, where \texttt{exit} is a variable defined ad hoc and it is set to \texttt{FALSE} by any transition that “goes out” of the cycle; (iii) connect the overall \texttt{<while>} block to other state skeletons by appropriate \texttt{<link>}s.

There are some interesting special cases: (i) a state $S$ with self-transitions can be represented as a \texttt{<pick>} block enclosed in a \texttt{<while>} block controlled by a condition ($Vs$) (the variable $Vs$ is set to \texttt{FALSE} by other non self-transitions);
(ii) cycles starting from the initial state should not be considered, as they can be represented as the start of a new instance of the BPEL4WS process.

By remarking the fact that the Realization Module is still in the development phase, we present in Figure 14 the BPEL4WS pseudo code for the MFSM of the running example.
Figure 14: BPEL4WS pseudo-code for the MFSM shown in Figure 9. The detail of a transition skeleton is shown only for one operation.
6. Related Work

Service Oriented Computing promises to give rise to new opportunities in developing and deploying distributed software applications, by suitably assembling services offered by different organizations. This is facilitated by the use of open (XML-based) standard languages (e.g., WSDL, WSCL, WSCI, BPEL4WS, WS-CDL) and protocols (such as SOAP and XML Protocol), which provide a basic substrate for wiring together the different services constituting the distributed application.

However, such standards lack a clear formal semantics, and therefore, they are not suitable for service oriented computing at a conceptual level.

Indeed, service oriented computing should be based on a conceptual representation of services from an external point of view, thus abstracting from internal (i.e., implementation) details; such an external point of view is the one to be considered when composing and orchestrating services. In this paper we have proposed a conceptual way of representing service behavior as finite state machines, in terms of both the internal and the external view, which constitutes an abstraction over current standards and technologies. On the basis of such a description we have developed a novel technique for automatic service composition.

Supported by such a technological layer, research on service oriented computing has mainly concentrated on (i) service description and modeling (i.e., what properties of a service should be described, and at which abstraction level), (ii) service discovery (i.e., how to efficiently query against service descriptions), (iii) service composition (i.e., how to specify goals and constraints of a composition, how to build a composition, how to analyze a composition), and (iv) orchestration (i.e., invocation, enactment and monitoring of both simple and composite services).

Service Description and Modeling. The OWL-S (formerly DAML-S) Coalition defines a specific ontology and a related language for services. A service presents a Service Profile (i.e., what it does, in terms of inputs and outputs, pre-conditions and effects), it is described by a Service Process Model (i.e., how it works, in terms of the abstract internal process), and it supports a Service Grounding (i.e., how to access the service, in terms of communication protocol, marshalling and serialization, etc.). Services, whose process is characterized by a FSM-based conceptual model and is described in OWL-S, can be easily composed using our technique.

In Bultan et al., 2003, a service is modeled as a Mealy machine, with input and output messages, and a (bounded) queue is used to buffer messages that are received but not yet processed. In our paper, we model services as finite state machines, but we do not consider communication delays and therefore any concept of message queuing is not taken into account. However, it is possible to show that the two models have the same expressive power. Moreover, from the survey of Hull et al., 2003, it stems that most practical and currently adopted approaches for modeling and describing services, which are targeted to composition, are based on finite automata/state machines.
Service Discovery. In van den Heuvel et al., 2001, services are considered as constituted by sub-services, thus modelled as a hierarchy of parts (expressing capabilities of services), based on a common ontology. On the assumption that all descriptions of available services are stored in a common repository, an algorithm that selects the service that best fits a given description (i.e., the request for specific capabilities) is presented, based on similarity notions. Such a selection is currently carried out only on the basis of static features similarity, whereas we argue that selection should also be based on behavioral descriptions.

Other works on service discovery propose information retrieval techniques, peer-to-peer scenarios and graph-based techniques in the context of OWL-S services. Although our work is orthogonal to service discovery issues, we would like to remark that all such approaches take into account only “static” service signatures, whereas considering behavioral descriptions could improve the quality of the discovery process. In our work we assume that the service community has already been assembled. Therefore, service discovery techniques play an important role in such a community construction phase.

Service Composition. In Yang and Papazoglou, 2004, a methodological framework for service composition and life-cycle management is proposed, in which composite services are created by re-using, specializing and extending existing ones.

In McIlraith and Son, 2002 and in McIlraith et al., 2001, a Situation Calculus based framework for services is proposed, where a service is described from the client point of view, as an atomic action, thus presenting an input/output behavior; a situation tree (i.e., a kind of process flow in the theory of Situation Calculus) is associated with such an atomic action. Services are specified as ConGolog procedures and a tool for automatic composition is presented: a user presents his goal to the system, expressed as a kind of generic (i.e., skeleton) procedure with user constraints and preferences. Such a user specification cannot be executed “as is”: it should be made executable by an agent that, exploiting a OWL-S ontology of services, automatically instantiates the user specification with services contained in such an ontology, by possibly pruning the situation tree corresponding to the generic procedure in order to take user preferences and constraints into account. Such an instantiated user specification is a sequence of atomic actions (i.e., services) which are then executed by a ConGolog interpreter. The main difference with our technique is that services are seen as atomic, therefore the client can not specify the interleaved execution of “pieces of” services (i.e., parts of atomic actions/procedures). Another difference is that in McIlraith and Son, 2002 and in McIlraith et al., 2001 the client specifies his goal once and forall before the composition and during the execution of the composite services he has no control on the executed sequences of actions. Conversely, in our work the client has such control, since at each step of the execution he chooses the next action to perform.
Finally, in McIlraith et al., 2001 the outcome of the composition is not a service, in the sense that it cannot be re-used by another client, whereas in our work the composition produces a reusable specification.

In Bultan et al., 2003, a framework for modeling and analyzing the global behavior of service compositions is presented. Services exchange messages according to a predefined communication topology, expressed as a set of channels among services: a sequence of exchanged messages (as seen by an external virtual watcher) is referred to as conversation. In this framework properties of conversations are studied, in order to characterize the behavior of services, modeled as Mealy machines. In such a framework, the synthesis problem takes in input (i) a desired global behavior (i.e., the set of all possible desired conversations) specified as a Linear Temporal Logic (LTL) formula, and (ii) a composition infrastructure, that is a set of channels, a set of (name of) services and a set of messages. The output of the synthesis is the specification of the Mealy machines of the services such that their conversations are compliant with the LTL specification. The main difference with our technique is that their approach to the synthesis is “top-down”: a desired global behavior is specified, and it is assumed that services can be designed during the synthesis phase without constraints. Conversely, our technique is “bottom/up”: the behavior of the services is also given, and the synthesis phase tries to assemble such behaviors in order to provide the desired behavior. Another difference consists in the conceptual model underlying the desired composition specification: in Bultan et al., 2003 a linear setting is taken, since composition focuses on linear sequences (i.e., paths) of actions; conversely, in our approach the client specification is based on a branching model: composition focuses on a tree-based structure, where each node denotes a choice point on what to do next. The expressive power of linear and branching temporal formulas is not comparable.

In Aiello et al., 2002, Lazovik et al., 2003, Pistore et al. 2004, Traverso and Pistore, 2004 a way of composing services is presented, based on planning under uncertainty, model checking and constraint satisfaction techniques, and a request language, to be used for specifying client goals, is proposed. Specifically, Lazovik et al., 2003 present an approach to service composition of atomic services based on interleaving of planning, monitoring and execution: in this way, the authors are able to adapt at runtime the composite service generated during the planning phase, to cope with possible changes in the service environment. Pistore et al. 2004, Traverso and Pistore, 2004 present a composition algorithm, that takes in input a set of partially specified services, modeled as non-deterministic finite state machines, and the client goals expressed as a branching temporal formula, and returns a plan that specifies how to coordinate the execution of concurrent services in order to realize the client goal. The plan can then be encoded in standard coordination languages and executed by orchestration engines. Note that, differently from our

*We want to recall that throughout the paper we use the term “client” to denote in the abstract the entity which is interested in having a composite service.
approach, once the plan is synthesized, the client has no further control on the execution of the service, in order to choose what to do next. The plan is also able to monitor the composition, thus guaranteeing that the interactions among available services satisfy given properties. Such proposals have some similarities with ours: indeed, in both cases the client goal essentially specifies temporal properties that the (behavior of the) overall composition should satisfy.

The results in this paper show that two specific features form the base of our proposal. The first one is that the composition involves the concurrent executions of several services. Only few proposals in the literature follow a similar idea. In particular, the most related ones are \cite{20,45,46,49,54}: they have in common with our proposal the fact that the service execution can be interleaved if needed. The composition deals with suitably controlling such an interleaving so as to realize the client request. Note that, most work on composition involves reaching a situation where some desired properties hold, and it is based on the idea of sequentially composing the available services, which are considered as black boxes, and hence atomically executed, as in \cite{4,40,44,60,1,42}. Such an approach to composition is tightly related to Classical Planning in AI \cite{30}. The second basic feature is that the client request is a specification of the transition system that the client wants to be able to execute. This feature is, to the best of our knowledge, unique to our proposal. Indeed even \cite{20,45,46} actually focus on realizing a single execution fulfilling the client request. Notice that such an execution may depend on conditions to be verified at run time, but not on further choices made by the client. Only the proposal in \cite{49,54} has some similarities with ours: indeed, there, the client goal is expressed in a specific branching-time logic, that allows to specify alternative paths of execution, which are, however, not under the control of the client, as it is in our case. However, their goals are still essentially based on having a main execution to follow, plus some side paths that are typically used to resolve exceptional circumstances.

Another point we want to discuss here regards the distinction between data and process that often shows up in the service literature. Indeed we have two extremes in dealing with data and process. One end of the spectrum is well explored by the literature on data integration that fully takes into account the data, but not the process \cite{34,43}. Interestingly, there are some proposals that base service composition for data intensive services on such a literature, avoiding to talk about the process as much as possible \cite{31}. The other end of the spectrum is much less studied. Our proposal, together with those in \cite{20,45,46,49,54}, tries to explore such an end of the spectrum. Observe that, introducing data in a naive way in our setting is in fact possible, but would make composition exponential in the data. This is to be considered unacceptable, since the amount of data is typically huge (wrt the size of the services) and hence one wants to keep the computations polynomial in the data. More generally, both ends of this spectrum (only data and only process) deal with problems that are quite difficult. Finding a good way to integrate the two, without multiplying the complexities, is probably going to become one of the key
problems in service composition in the future.

Finally, related to service composition is the analysis and verification of composite services, motivated by the dynamic flavor of composition, the consequent difficulties in testing and immaturity of service oriented development environments and methods. Preliminary results can be found in Fu et al., 2004, where verification of BPEL4WS specifications is carried out exploiting model checking techniques, in Narayanan and McIlraith, 2002, where OWL-S services are analyzed exploiting Petri Nets, and in Deutsch et al., 2004, which focuses on verifying properties of data-driven services. Finally, we want to remark that analysis and verification are more effective when composite services are manually synthesized; our technique automatically synthesize a composite service that is correct by construction according to Section 4.

Orchestration. Orchestration requires that the composite service is completely specified, in terms of both the specification of how various component services are linked, and the internal process flow of the composite one. In Hull et al., 2003, different technologies, standards and approaches for specification of composite services are considered, including BPEL4WS, BPML, AZTEC, etc. In particular, Hull et al., 2003 identifies two main kinds of composition: (i) the mediated approach, based on a hub-and-spoke topology, in which one service is given the role of process mediator/delegator, and all the interactions pass through such a service, and (ii) the peer-to-peer approach, in which the services directly interact among them, without any centralized control. With respect to such a classification, the approach proposed in this paper belongs to the mediated one.

Many orchestration platforms have been designed and proposed in the literature (e.g., e-Flow, AZTEC, WISE, MENTOR-lite, e-ADOME): they can be classified into the mediated approach to composition. An interesting case is Self-Serv, in which the enactment of a composite service (to be manually designed) is carried out in a decentralized way, through peer-to-peer interactions.

Finally, we would like to remark that our results are orthogonal to service orchestration; once the Mealy composition, obtained with our technique, is translated into a specific orchestration language (in the paper we have discussed the case of BPEL4WS), the obtained specification can be orchestrated by any orchestration platform, thus obtaining all system-level guarantees needed in complex distributed applications.

7. Final Remarks and Future Work

The main contribution of this paper wrt research on service oriented computing is in tackling simultaneously the following issues: (i) presenting a formal framework where services are characterized in terms of their behavioral descriptions and the problem of service composition is precisely defined; (ii) providing techniques for computing service composition in the case where the behavioral description of ser-
Automatic Service Composition based on Behavioral Descriptions

Services is expressed as finite state machine, and providing a computational complexity characterization of the algorithm for automatic composition; (iii) presenting \( \text{ESC} \), an open source prototype tool that implements our technique for automatically synthesizing a composition.

In \(^{18}\) we have extended our framework by allowing some advanced forms of non-determinism in the client request, which can be loosely specified, and we devised automatic composition techniques in this enhanced framework. In the future, we plan to produce a new version of our prototype tool that takes such extensions into account.

Currently, we are investigating how to add data in our framework in a “smart” way, by taking into account the considerations made in Section 6, so that the combination of data and process allows us to devise algorithms for automatic service composition with reasonable computational complexity.

Also, we aim at establishing a lower bound characterization for the problem of service composition.

Finally, far-reaching future work may be identified along several directions. For example, it could be interesting to study the situation when the available services export a partial description of their behavior, i.e., they are represented by non-deterministic FSMs. This means that a large (possibly infinite) number of complete description for services in the community exists that are coherent with each partial description. Note that the internal schema to be synthesized should be coherent with all such possible complete descriptions. Therefore, computing composition in such a framework is intuitively much more difficult than in the framework presented here.

Acknowledgments

This work has been supported by MIUR through the “FIRB 2001” project MAIS (http://www.mais-project.it, Workpackage 2), and “Società dell’Informazione” sub-project SP1 “Reti Internet: Efficienza, Integrazione e Sicurezza”. It has been also supported by the European projects SEWASIE (IST-2001-34825), EU-PUBLI.com (IST-2001-35217) and INTEROP Network of Excellence (IST-508011). The authors would like to thank Richard Hull, for useful comments and discussion, Alessandro Iuliani, for collaborating in the design and realization of the \( \text{ESC} \) tool, and Alessia Candido for her technical support with BPEL4WS. Finally, the authors would like to thank the anonymous reviewers for valuable suggestions that helped improving the paper.

References


56. W.J. van den Heuvel, J. Yang, and M.P. Papazoglou. Service Representation, Discovery
Automatic Service Composition based on Behavioral Descriptions


