International Journal of Cooperative Information Systems
Vol. 14, No. 4 (2005) 333–376

World Scientific
www.worldscientific.com

AUTOMATIC SERVICE COMPOSITION BASED ON BEHAVIORAL DESCRIPTIONS

DANIELA BERARDI*,†, DIEGO CALVANESE‡, GIUSEPPE DE GIACOMO†,
MAURIZIO LENZERINI† and MASSIMO MECELLA†

†*Dipartimento di Informatica e Sistemistica "A. Ruberti"*
Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italy
**berardi@dis.uniroma1.it*

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

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 $\mathcal{ESC}$ (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[2,49]) aims at building agile networks of collaborating business applications, distributed within and across organizational boundaries. 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, published, located and dynamically invoked, in a programming language independent way.

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

*Corresponding author.

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 w.r.t. its components, since it (indirectly) looks for and invokes them. Service composition leads to enhancements of the SOA (Extended SOA[49]), 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 on 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[38]), but an agreed-upon comprehension of what a 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[a] can choose the next action 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.

[a]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.

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.[2,12,38] 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 of 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 open to the exact lower bound for the problem, 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 a third contribution to the paper, we present the prototype design and development of an open source software tool implementing our composition technique, namely $\mathcal{ESC}$ (E-Service Composer).[b] 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 Sec. 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 Sec. 3, we exploit the general framework to study the case where services can be characterized by a finite number of states. In Sec. 4, we present a sound, complete and terminating technique for the automatic synthesis of composition. In Sec. 5, we present our prototype tool $\mathcal{ESC}$. Finally, in Sec. 6, we consider related research work and in Sec. 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 either be a human user, or another service. When executed, a service performs its task by directly 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 lifetime.

- 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.

[b]cf. the PARIDE (Process-based frAmeworRk 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.

- 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 Refs. 14 and 17.

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.[c] 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

[c]We assume for simplicity that the list of songs returned by `search_by_author` and `search_by_title` is non-empty.

can be found in standard languages such as WSDL[24] 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 Σ, 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 Σ. 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 Σ. From a more practical point of view, a community can be seen 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[9] or tightly-coupled *e*-Business[26] 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 *composite*. If this is not the case, a service is called *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 his 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.

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 *exported behavior* represented as trees of atomic *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 *external schema* and *internal schema*, respectively.

Accordingly, service instances are characterized by an external and an internal view.[17]

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[d] that has executed the path to x. For every action a belonging to the alphabet Σ 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

[d]In what follows, we omit the terms "schema" and "instance" when clear from the context.

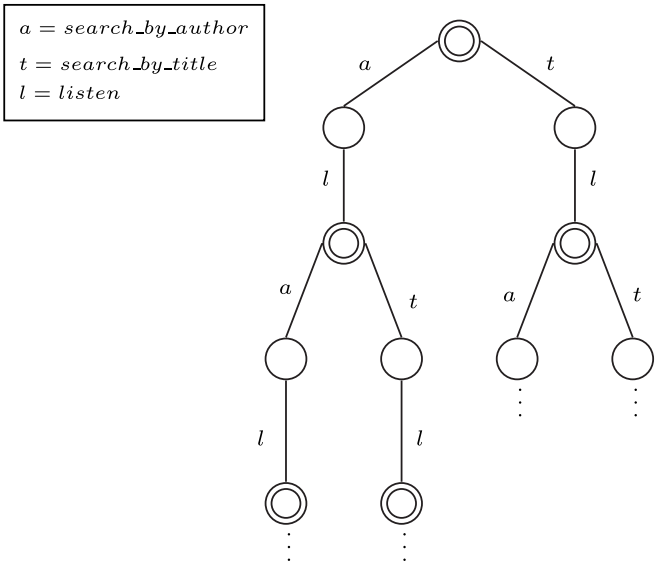


Fig. 1. External execution tree of service E_0.

tree and we say that such an edge is *labeled* with action a. The root ε 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.[e]

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 $T(E^{\text{ext}})$ 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.[f] 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).

[e]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.
[f]Final nodes are represented by two concentric circles.

2.2.2. *Internal schema*

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 `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_{int} we call *offered external execution tree* the external execution tree T_{ext} obtained from T_{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_{int} *conforms to* an external execution tree T_{ext} if T_{ext} is equal to the offered external execution tree of T_{int}.

Given a service E, the internal schema E^{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^{ext} and internal schema E^{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

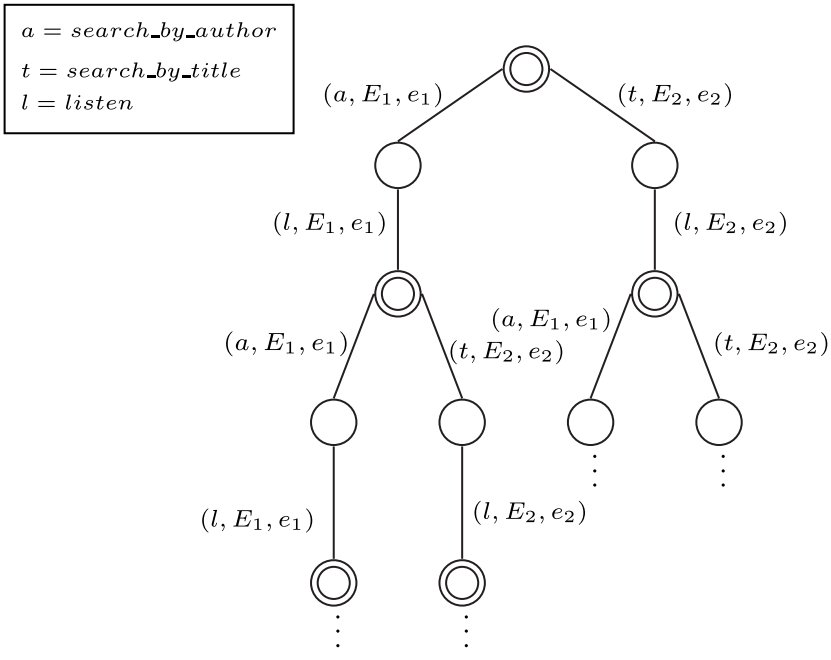


Fig. 2. Internal execution tree of service E_0.

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,[g] conforming to the external execution tree of service E_0 shown in Fig. 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.

[g]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 $i = 1, 2$.

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 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^{ext} be the external schema of a target service E expressed in terms of the alphabet Σ of C. A *composition* of E w.r.t. C is an internal schema E^{int} such that:

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

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

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

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

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.[18,21] 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.

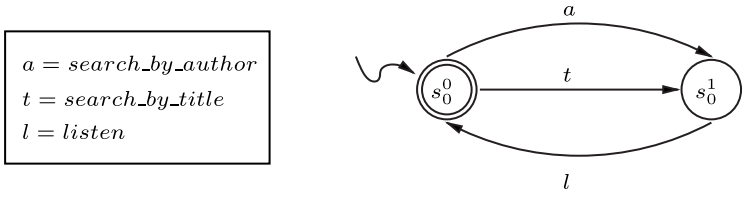
Definition 8 [(FSM) External Schema]. Let E be a service. The external schema of E is a FSM $A_E^{\text{ext}} = (\Sigma, S_E, s_E^0, \delta_E, F_E)$, where:

- Σ 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_E^0 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.

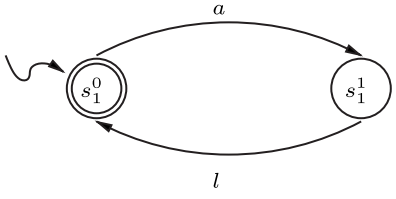
Example 4. Figure 3(a) shows the external schema of the target service E_0 of Examples 2 and 3, specified by the client as a FSM A_0. Figures 3(b) and (c) show the external schemas, represented as FSMs A_1 and A_2, respectively associated to component services E_1 and E_2 of Example 3. In other words, 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 `search_by_author`) and for listening to the song selected by the client (action `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 `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 $\Sigma = \{\texttt{search\_by\_author}, \texttt{search\_by\_title}, \texttt{listen}\}$. According to our setting, the client specifies the external schema A_0 of his target service in terms of Σ.

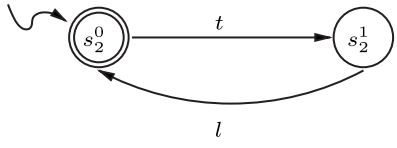
The FSM A_E^{ext} is an external schema in the sense that it *specifies* an external execution tree $T(A_E^{\text{ext}})$. Specifically, given A_E^{ext} we define $T(A_E^{\text{ext}})$ inductively on the level of nodes in the tree, by making use of an auxiliary function $\sigma(\cdot)$ that associates



(a) External schema A_0 of the target service E_0.



(b) External schema A_1 of the component service E_1.



(c) External schema A_2 of the component service E_2.

Fig. 3. Composition of services.

to each node of the tree a state in the FSM. We proceed as follows:

- ε, as usual, is the root of $T(A_E^{\text{ext}})$ and $\sigma(\varepsilon) = s_E^0$;
- if x is a node of $T(A_E^{\text{ext}})$, 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 $T(A_E^{\text{ext}})$ and $\sigma(x \cdot a) = s'$;
- x is final iff $\sigma(x) \in F_E$.

Example 5. Figure 4 shows (a portion of the) the external execution tree $T(A_0)$ defined from A_0 by a mapping σ (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 σ associates to it. The mapping σ is defined as follows.

$$\begin{aligned}
&\sigma(\varepsilon) = s_0^0\\
&\sigma(a) = \sigma(t) = s_0^1\\
&\sigma(a \cdot l) = \sigma(t \cdot l) = s_0^0\\
&\sigma(a \cdot l \cdot a) = \sigma(a \cdot l \cdot t) = \sigma(t \cdot l \cdot a) = \sigma(t \cdot l \cdot t) = s_0^1\\
&\sigma(a \cdot l \cdot a \cdot l) = \sigma(a \cdot l \cdot t \cdot l) = \sigma(t \cdot l \cdot a \cdot l) = \sigma(t \cdot l \cdot t \cdot l) = s_0^0\\
&\ldots
\end{aligned}$$

σ 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_{ext} of Fig. 1. That is, T_{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.

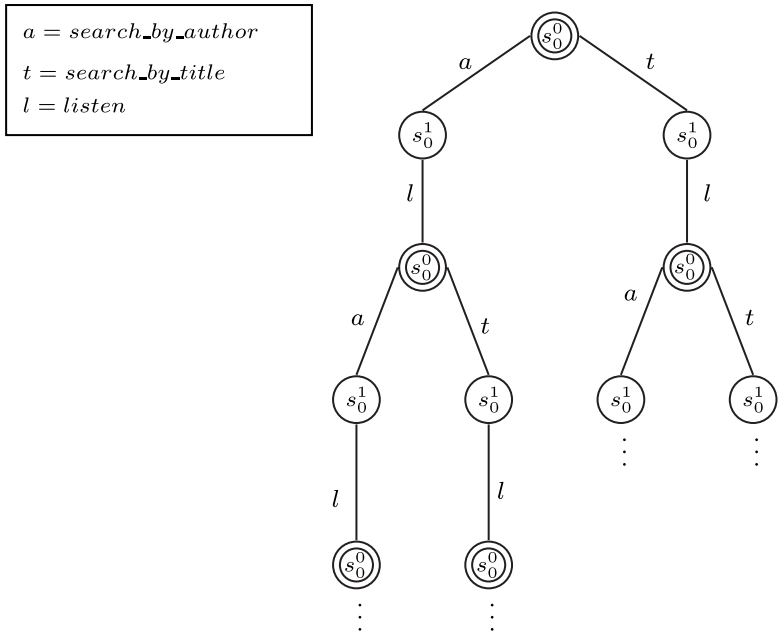


Fig. 4. External execution tree $T(A_0)$.

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) $A_E^{\text{int}} = (\Sigma, 2^{[n]}, S_E^{\text{int}}, s_E^{0\,\text{int}}, \delta_E^{\text{int}}, \omega_E^{\text{int}}, F_E^{\text{int}})$, where:

- $\Sigma, S_E^{\text{int}}, s_E^{0\,\text{int}}, \delta_E^{\text{int}}, F_E^{\text{int}}$, have the same meaning as for A_E^{ext};
- $2^{[n]}$ is the output alphabet of the MFSM, which is used to denote which service(s) executes each action;
- $\omega_E^{\text{int}} : S_E^{\text{int}} \times \Sigma \to 2^{[n]}$ 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 `this` is implied; we assume that the output function ω_E^{int} is defined exactly when δ_E^{int} is so.

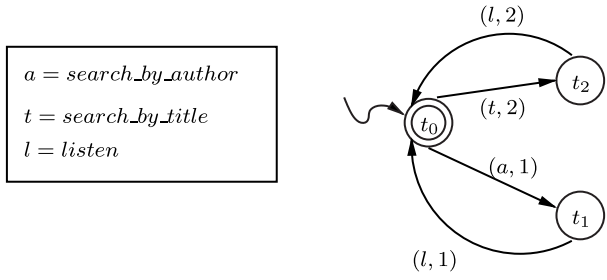


Fig. 5. Service internal specification as MFSM M_0.

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 ω^{int} is defined as follows:

$$\omega^{\text{int}}(s_0^0, a) = \{1\} \quad \omega^{\text{int}}(s_0^0, t) = \{2\}$$
$$\omega^{\text{int}}(s_0^1, l) = \{1\} \quad \omega^{\text{int}}(s_0^2, l) = \{2\}.$$

The MFSM A_E^{int} is an internal schema in the sense that it specifies an internal execution tree $T(A_E^{\text{int}})$. Given A_E^{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:

- ε is, as usual, the root of $T(A_E^{\text{int}})$ and $\sigma^{\text{int}}(\varepsilon) = s_E^{0\,\text{int}}$;
- if x is a node of $T(A_E^{\text{int}})$, and $\sigma^{\text{int}}(x) = s$, for some $s \in S_E^{\text{int}}$, then for each a such that $s' = \delta_E^{\text{int}}(s, a)$ is defined, $x \cdot a$ is a node of $T(A_E^{\text{int}})$ and $\sigma^{\text{int}}(x \cdot a) = s'$;
- if x is a node of $T(A_E^{\text{int}})$, and $\sigma^{\text{int}}(x) = s$, for some $s \in S_E^{\text{int}}$, then for each a such that $\omega_E^{\text{int}}(s, a)$ is defined (i.e. $\delta_E^{\text{int}}(s, a)$ is defined), the edge $(x, x \cdot a)$ of the tree is labeled by $\omega_E^{\text{int}}(s, a)$;
- x is final iff $\sigma^{\text{int}}(x) \in F_E^{\text{int}}$.

Example 7. Figure 6 shows a portion of the internal execution tree $T(M_0)$ defined from M_0, shown in Fig. 5. Each node of the tree is labeled with the state of M_0 that σ^{int} associates to it. The mapping σ^{int} is defined as follows.

$$\sigma^{\text{int}}(\varepsilon) = s_0^0$$
$$\sigma^{\text{int}}(a) = s_0^1$$
$$\sigma^{\text{int}}(t) = s_0^2$$
$$\sigma^{\text{int}}(a \cdot l) = \sigma^{\text{int}}(t \cdot l) = s_0^0$$
$$\sigma^{\text{int}}(a \cdot l \cdot a) = \sigma^{\text{int}}(t \cdot l \cdot a) = s_0^1$$
$$\sigma^{\text{int}}(a \cdot l \cdot t) = \sigma^{\text{int}}(t \cdot l \cdot t) = s_0^2$$
$$\sigma^{\text{int}}(a \cdot l \cdot a \cdot l) = \sigma^{\text{int}}(a \cdot l \cdot t \cdot l) = \sigma^{\text{int}}(t \cdot l \cdot a \cdot l) = \sigma^{\text{int}}(t \cdot l \cdot t \cdot l) = s_0^0$$
$$\ldots$$

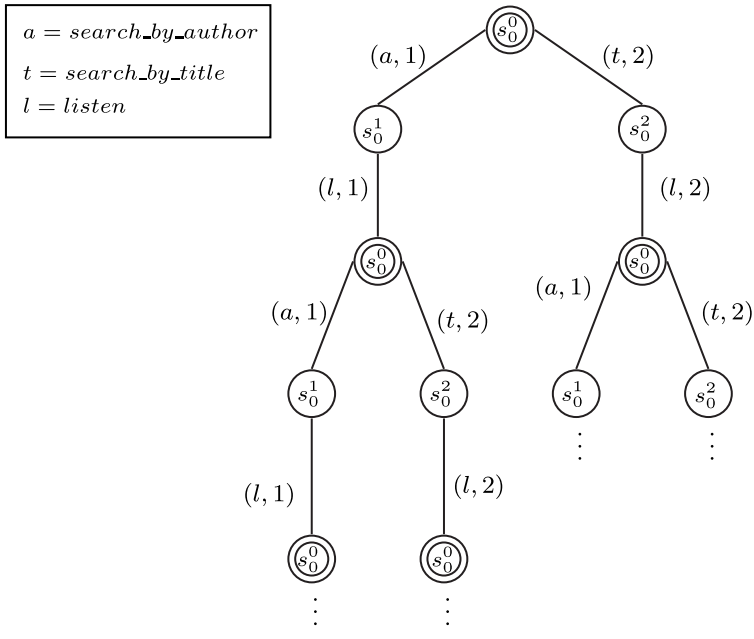


Fig. 6. Internal execution tree $T(M_0)$.

σ^{int} maps over s_0^1 the nodes of the tree that represent strings ending by a, and over s_0^2 the nodes that represent strings ending by t; it maps over s_0^0 the root and the nodes of the tree associated to strings ending by l.

Note that $T(M_0)$ is equal to the internal execution tree T_{int} of Fig. 2 (up to renaming the labels (E_i, e_i) with i). That is, T_{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 Fig. 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. 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.[40]

4.1. *Deterministic propositional dynamic logic*

Propositional Dynamic Logics (PDLs) are a family of modal logics specifically developed for reasoning about computer programs.[40] 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.[36]

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

$$\phi \rightarrow \texttt{true} \mid \texttt{false} \mid P \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \langle r \rangle \phi \mid [r]\phi$$
$$r \rightarrow a \mid r_1 \cup r_2 \mid r_1; r_2 | r^* | \phi?$$

where P is an atomic proposition in $\mathcal{P}$, r is a regular expression over the set of actions in $\mathcal{A}$, and a is an atomic action in $\mathcal{A}$. 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 ϕ holds, and that all terminating executions of r reach a state where ϕ 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 r_1 then execute r_2"; r^* means "execute r a non-deterministically chosen number of times (zero or more)"; $\phi?$ means "test ϕ: if it is true proceed else fail".

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 $\mathcal{I} = (\Delta^{\mathcal{I}}, \{a^{\mathcal{I}}\}_{a \in \mathcal{A}}, \{P^{\mathcal{I}}\}_{P \in \mathcal{P}})$, where $\Delta^{\mathcal{I}}$ denotes a non-empty set of states (also called worlds); $\{a^{\mathcal{I}}\}_{a \in \mathcal{A}}$ is a family of partial *functions* $a^{\mathcal{I}} : \Delta^{\mathcal{I}} \rightarrow \Delta^{\mathcal{I}}$ from elements of $\Delta^{\mathcal{I}}$ to elements of $\Delta^{\mathcal{I}}$, each of which denotes the state transition caused by the atomic program a; $P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ denotes all the elements of $\Delta^{\mathcal{I}}$ were P is true.

The semantic relation "a formula ϕ holds at a state s of a structure $\mathcal{I}$", is written $\mathcal{I}, s \models \phi$, and is defined by induction on the form of ϕ:

$$
\begin{array}{ll}
\mathcal{I}, s \models \texttt{true} & \text{always} \\
\mathcal{I}, s \models \texttt{false} & \text{never} \\
\mathcal{I}, s \models P & \text{iff } s \in P^{\mathcal{I}} \\
\mathcal{I}, s \models \neg\phi & \text{iff } \mathcal{I}, s \not\models \phi \\
\mathcal{I}, s \models \phi_1 \wedge \phi_2 & \text{iff } \mathcal{I}, s \models \phi_1 \quad \text{and} \quad \mathcal{I}, s \models \phi_2 \\
\mathcal{I}, s \models \phi_1 \vee \phi_2 & \text{iff } \mathcal{I}, s \models \phi_1 \quad \text{or} \quad \mathcal{I}, s \models \phi_2 \\
\mathcal{I}, s \models \langle r \rangle \phi & \text{iff there is } s' \text{ such that } (s, s') \in r^{\mathcal{I}} \quad \text{and} \quad \mathcal{I}, s' \models \phi \\
\mathcal{I}, s \models [r]\phi & \text{iff for all } s', (s, s') \in r^{\mathcal{I}} \text{ implies } \mathcal{I}, s' \models \phi
\end{array}
$$

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

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

It is important to understand, given a formula ϕ, which are the formulas that play some role in establishing the truth-value of ϕ. In simpler modal logics, these formulas are simply all the subformulas of ϕ, 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.[29]

A structure $\mathcal{I} = (\Delta^{\mathcal{I}}, \{a^{\mathcal{I}}\}_{a \in \mathcal{A}}, \{P^{\mathcal{I}}\}_{P \in \mathcal{P}})$ is called a *model* of a formula ϕ if there exists a state $s \in \Delta^{\mathcal{I}}$ such that $\mathcal{I}, s \models \phi$. A formula ϕ is *satisfiable* if there exists a model of ϕ, otherwise the formula is *unsatisfiable*. A formula ϕ is *valid* in structure $\mathcal{I}$ if for all $s \in \Delta^{\mathcal{I}}$, $\mathcal{I}, s \models \phi$. We call *axioms* formulas that are used to select the interpretations of interest. Formally, a structure $\mathcal{I}$ is a model of an axiom ϕ, if ϕ is valid in $\mathcal{I}$. A structure $\mathcal{I}$ is a model of a finite set of axioms Γ if $\mathcal{I}$ is a model of all axioms in Γ. 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 Γ of axioms *logically implies* a formula ϕ, written $\Gamma \models \phi$, if ϕ is valid in every model of Γ. It is easy to see that satisfiability of a formula ϕ as well as satisfiability of a finite set of axioms Γ 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:[10]

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 Φ as follows. As set of atomic propositions $\mathcal{P}$ in Φ we have (i) one proposition s_j for each state s_j of $A_j, j = 0, \ldots, n$, denoting whether A_j is in state s_j; (ii) propositions $F_j, j = 0, \ldots, n$, denoting whether A_j is in a final state; and (iii) propositions $moved_j, j = 1, \ldots, n$, denoting whether (component) FSM A_j performed a transition. As set of atomic actions $\mathcal{A}$ in Φ we have the actions in Σ (i.e. $\mathcal{A} = \Sigma$).

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

$$\mathcal{P} = \{s_0^0, s_0^1, s_1^0, s_1^1, s_2^0, s_2^1, F_0, F_1, F_2, moved_1, moved_2\}$$

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;
- $moved_j$ for $j = 1, 2$: (component) FSM A_j performed a transition.

The set $\mathcal{A}$ of deterministic atomic actions, which by construction coincides with the alphabet of the community, is defined as follows:

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

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 Φ is built as a conjunction of the following formulas.

- Formulas representing $A_0 = (\Sigma, S_0, s_0^0, \delta_0, F_0)$:
  - $[u](s \rightarrow \neg s')$ 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).
  - $[u](s \rightarrow \langle a \rangle \mathtt{true} \wedge [a]s')$ for each a such that $s' = \delta_0(s, a)$; these encode the transitions of A_0.

— $[u](s \rightarrow [a]\texttt{false})$ 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 $\mathcal{A}$. 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 ϕ is a proposition, asserts that ϕ 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 $[u](s_0^1 \rightarrow \neg s_0^0)$.

$$[u](s_0^0 \rightarrow \langle a \rangle \texttt{true} \wedge [a]s_0^1)$$
$$[u](s_0^0 \rightarrow \langle t \rangle \texttt{true} \wedge [t]s_0^1)$$
$$[u](s_0^1 \rightarrow \langle l \rangle \texttt{true} \wedge [l]s_0^0).$$

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_0^1. Analogously for the other formulas.

$$[u](s_0^0 \rightarrow [l]\texttt{false})$$
$$[u](s_0^1 \rightarrow [a]\texttt{false})$$
$$[u](s_0^1 \rightarrow [t]\texttt{false}).$$

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 `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.
— $[u](s \rightarrow [a](\mathit{moved}_i \wedge s' \vee \neg \mathit{moved}_i \wedge s))$ 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.
— $[u](s \rightarrow [a](\neg \mathit{moved}_i \wedge s))$ 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.

$$[u](s_1^0 \rightarrow \neg s_1^1).$$

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

$$[u](s_1^0 \rightarrow [a](moved_1 \wedge s_1^1 \vee \neg moved_1 \wedge s_1^0))$$
$$[u](s_1^1 \rightarrow [l](moved_1 \wedge s_1^0 \vee \neg moved_1 \wedge s_1^1)).$$

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_1^0.

$$[u](s_1^0 \rightarrow [l](\neg moved_1 \wedge s_1^0))$$
$$[u](s_1^0 \rightarrow [t](\neg moved_1 \wedge s_1^0))$$
$$[u](s_1^1 \rightarrow [a](\neg moved_1 \wedge s_1^1))$$
$$[u](s_1^1 \rightarrow [t](\neg moved_1 \wedge s_1^1)).$$

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_1^0 and it receives actions l in input, it does not move, and therefore it remains in state s_1^0; 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

$$[u](F_1 \leftrightarrow s_1^0)$$

asserts that state s_1^0 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.

$$[u](s_2^0 \rightarrow \neg s_2^1)$$
$$[u](s_2^0 \rightarrow [t](moved_2 \wedge s_2^1 \vee \neg moved_2 \wedge s_2^0))$$
$$[u](s_2^1 \rightarrow [l](moved_2 \wedge s_2^0 \vee \neg moved_2 \wedge s_2^1))$$
$$[u](s_2^0 \rightarrow [l](\neg moved_2 \wedge s_2^0))$$
$$[u](s_2^0 \rightarrow [a](\neg moved_2 \wedge s_2^0))$$
$$[u](s_2^1 \rightarrow [t](\neg moved_2 \wedge s_2^1))$$
$$[u](s_2^1 \rightarrow [a](\neg moved_2 \wedge s_2^1))$$
$$[u](F_2 \leftrightarrow s_2^0).$$

- Finally, formulas encoding domain independent conditions:
  - $s_0^0 \wedge \bigwedge_{i=1,\ldots,n} s_i^0$; this says that initially all services are in their initial state; note that this formula is not prefixed by $[u](\cdot)$.
  - $[u](\langle a\rangle \mathtt{true} \rightarrow [a] \bigvee_{i=1,\ldots,n} moved_i)$, for each $a \in \Sigma$; these say that at each step at least one of the component FSM has moved.
  - $[u](F_0 \rightarrow \bigwedge_{i=1,\ldots,n} F_i)$; 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.

$$s_0^0 \wedge s_1^0 \wedge s_2^0.$$

It asserts that all services start from their initial states.

$$\begin{aligned}
&[u](\langle a\rangle \mathtt{true} \rightarrow [a](moved_1 \vee moved_2))\\
&[u](\langle t\rangle \mathtt{true} \rightarrow [t](moved_1 \vee moved_2))\\
&[u](\langle l\rangle \mathtt{true} \rightarrow [l](moved_1 \vee moved_2)).
\end{aligned}$$

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,

$$[u](F_0 \rightarrow F_1 \wedge F_2)$$

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 w.r.t. $E_1, \ldots, E_n$, then the DPDL formula Φ, 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 w.r.t. $E_1, \ldots, E_n$. 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 σ and σ_i from nodes in T_{int} to states of A_0 and A_i, respectively, by induction on the level of the nodes in T_{int} as follows:

- base case: $\sigma(\varepsilon) = s_0^0$ and $\sigma_i(\varepsilon) = s_i^0$;
- inductive case: let $\sigma(x) = s$ and $\sigma_i(x) = s_i$, and let the node $x \cdot a$ be in T_{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_{int} is specified by a composition). Then we define

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

and

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

Once we have σ and σ_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 Φ as follows:

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

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

Lemma 2. *Any model of the DPDL formula* Φ, *constructed as above, denotes a composition of* E_0 *w.r.t.* $E_1, \ldots, E_n$.

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

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

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 w.r.t. $E_1, \ldots, E_n$. □

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

Proof. Straightforward, from Lemmas 1 and 2. □

Observe that the size of Φ 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 Φ 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 Φ is satisfiable then it admits a model, which is exactly the internal schema, i.e. the composition we want to synthesize. Conversely, if Φ 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 Φ one can always obtain a model which is at most exponential in the size of Φ. From such a model, one can extract an internal schema for E_0 that is a composition of E_0 w.r.t. $E_1, \ldots, E_n$, and which has the form of a MFSM.

Definition 10 (Mealy Composition). Given a finite model $\mathcal{I}_f = (\Delta^{\mathcal{I}_f}, \{a^{\mathcal{I}_f}\}_{a\in\Sigma}, \{P^{\mathcal{I}_f}\}_{P\in\mathcal{P}})$, we define *Mealy composition* an MFSM $A_c = (\Sigma, 2^{[n]}, S_c, s_c^0, \delta_c, \omega_c, F_c,)$, built as follows:

- $S_c = \Delta^{\mathcal{I}_f}$;
- $s_c^0 = d_0$ where $d_0 \in (s_0^0 \wedge \bigwedge_{i=1,\ldots,n} s_i^0)^{\mathcal{I}_f}$;
- $s' = \delta_c(s, a)$ iff $(s, s') \in a^{\mathcal{I}_f}$;
- $I = \omega_c(s, a)$ iff $(s, s') \in a^{\mathcal{I}_f}$ and for all $i \in I$, $s' \in moved_i^{\mathcal{I}_f}$ and for all $j \notin I$, $s' \notin moved_j^{\mathcal{I}_f}$;
- $F_c = F_0^{\mathcal{I}_f}$.

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

Theorem 4. *If there exists a composition of E_0 w.r.t. $E_1, \ldots, E_0$, 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 Φ constructed as above is satisfiable. In turn, if Φ is satisfiable, for the small-model property of DPDL there exists a model $\mathcal{I}_f$ of size at most exponential in Φ, and hence in A_0 and $A_1, \ldots, A_n$. From $\mathcal{I}_f$ 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

$T(A_0)$, (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 *w.r.t.* $E_1, \ldots, E_n$.

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[7,27] 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 `Extract_MFSM`, `line 13`), according to Definition 10. Intuitively, the

AUTOMATIC SERVICE COMPOSITION

```
 1  INPUT:  A_0              /* FSM external schema of target service */
 2          A_1 ... A_n      /* FSM external schema of services in the community */
 3
 4  OUTPUT: if (a composition of A_0 wrt A_1 ... A_n exists)
 5              then return a Mealy composition of A_0 wrt A_1 ... A_n
 6              else return nil
 7
 8  begin
 9   Φ := FSM2DPDL(A_0, A_1, ... A_n);
10   I_f := DPDLTableau(Φ);
11   if (I_f == nil)
12      then return nil
13      else A_c := Extract_MFSM(I_f);
14           C_min := Minimize(A_c);
15           return C_min;
16  end
```

Fig. 7. The algorithm for synthesizing Mealy composition.

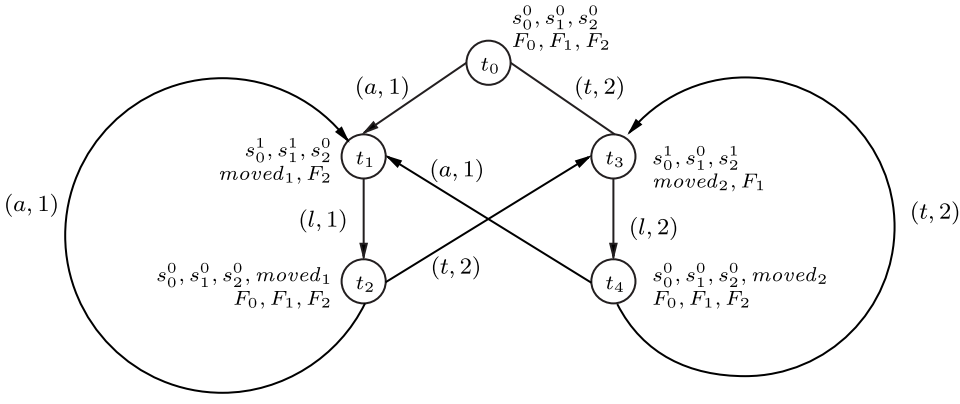


Fig. 8. Finite model $\mathcal{I}_f$ for Φ.

transformation from a finite model $\mathcal{I}_f$ to a Mealy Machine A_c consists in discarding from each state of $\mathcal{I}_f$ 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 `Minimize`), and the minimal Mealy composition $C_{\min}$ is returned (`line 15`). As we will show in Sec. 5, our prototype tool implements exactly such steps.

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

$$\Delta^{\mathcal{I}_f} = \{t_0, t_1, t_2, t_3, t_4\}$$
$$a^{\mathcal{I}_f} = \{(t_0, t_1), (t_2, t_1), (t_4, t_1)\}$$
$$t^{\mathcal{I}_f} = \{(t_0, t_3), (t_2, t_3), (t_4, t_3)\}$$
$$l^{\mathcal{I}_f} = \{(t_1, t_2), (t_3, t_4)\}$$
$$(s_0^0)^{\mathcal{I}_f} = \{t_0, t_2, t_4\}$$
$$(s_0^1)^{\mathcal{I}_f} = \{t_1, t_3\}$$
$$(s_1^0)^{\mathcal{I}_f} = \{t_0, t_2, t_3, t_4\}$$
$$(s_1^1)^{\mathcal{I}_f} = \{t_1\}$$
$$(s_2^0)^{\mathcal{I}_f} = \{t_0, t_1, t_2, t_4\}$$
$$(s_2^1)^{\mathcal{I}_f} = \{t_3\}$$
$$moved_1^{\mathcal{I}_f} = \{t_1, t_2\}$$
$$moved_2^{\mathcal{I}_f} = \{t_3, t_4\}$$
$$F_0^{\mathcal{I}_f} = \{t_0, t_2, t_4\}$$
$$F_1^{\mathcal{I}_f} = \{t_0, t_2, t_3, t_4\}$$
$$F_2^{\mathcal{I}_f} = \{t_0, t_1, t_2, t_4\}.$$

Each state t_i of the model is associated with the atomic propositions in $\mathcal{P}$ that hold in that state, according to $\mathcal{I}_f$. For example, consider state t_0 (which is initial for the model): $\mathcal{I}_f$ imposes that $s_0^0 \wedge s_1^0 \wedge s_2^0 \wedge F_0 \wedge F_1 \wedge F_2$ holds in t_0. For the sake of readability, in the figure we have associated to each state of $\mathcal{I}_f$ simply the list of atomic propositions that are true. Additionally, note that the DPDL encoding

does not pose any constraint on the value of $moved_i$ predicates in the initial state of the model: their value has been arbitrarily chosen to be `false`.[h]

Given $\mathcal{I}_f = (\Delta^{\mathcal{I}_f}, \{a^{\mathcal{I}_f}\}_{a\in\Sigma}, \{P^{\mathcal{I}_f}\}_{P\in\mathcal{P}})$ of Φ, we define a Mealy Machine $A_c = (\Sigma, 2^{[n]}, S_c, s_c^0, \delta_c, \omega_c, F_c,)$ representing the internal schema of the target service, as follows:

- $S_c = \{t_0, t_1, t_2, t_3, t_4\}$;
- $s_c^0 = t_0$, where $t_0 \in (s_0^0 \wedge s_1^0 \wedge s_2^0)^{\mathcal{I}_f}$; note that we could have as well as chosen either t_2 or t_4 as initial state;
- δ_c is defined as:

$$\begin{array}{ll} \delta_c(t_0, a) = t_1 & \delta_c(t_2, a) = t_1 \\ \delta_c(t_0, t) = t_3 & \delta_c(t_2, t) = t_3 \\ \delta_c(t_1, l) = t_2 & \delta_c(t_4, a) = t_1 \\ \delta_c(t_3, l) = t_4 & \delta_c(t_4, t) = t_3 \end{array}$$

- ω_c is defined as:

$$\begin{array}{ll} \omega_c(t_0, a) = \{1\} & \omega_c(t_2, a) = \{1\} \\ \omega_c(t_0, t) = \{2\} & \omega_c(t_2, t) = \{2\} \\ \omega_c(t_1, l) = \{1\} & \omega_c(t_4, a) = \{1\} \\ \omega_c(t_3, l) = \{2\} & \omega_c(t_4, t) = \{2\} \end{array}$$

- $F_c = \{t_0, t_2, t_4\}$.

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

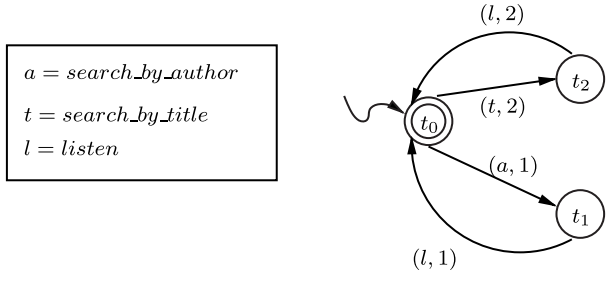


Fig. 9. Minimal MFSM $C_{\min}$ associated to $\mathcal{I}_f$.

[h]Note also the model for the DPDL formula Φ is deterministic, as it should be. Non-determinism could have been introduced by the operator $\langle\rangle$. However, we are guaranteed that no atomic action a connects state s_1 with two different target states s_2 and s_3, because $\langle\rangle$ appears only in front of the atomic proposition `true`. Indeed, if a related s_1 with s_2 and s_3, such target states would actually be the same, since s_2 and s_3 associated with the same atomic proposition `true`.

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 Fig. 7 can be extended as follows. It takes in input also a DPDL formula Φ_{Prop} encoding the additional constraints that the composition should satisfy. `Line 10` is replaced with $\mathcal{I}_f := \text{DPDLTableau}(\Phi \wedge \Phi_{\text{Prop}})$; satisfiability of the conjunct $\Phi \wedge \Phi_{\text{Prop}}$ is checked, and a model $\mathcal{I}_f$ is returned if one exists. It is easy to see that any model $\mathcal{I}_f$ 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,[i] which can be expressed in any language that allows to express a finite state machine (e.g. Web Service Conversation Language,[33] Web Service Transition

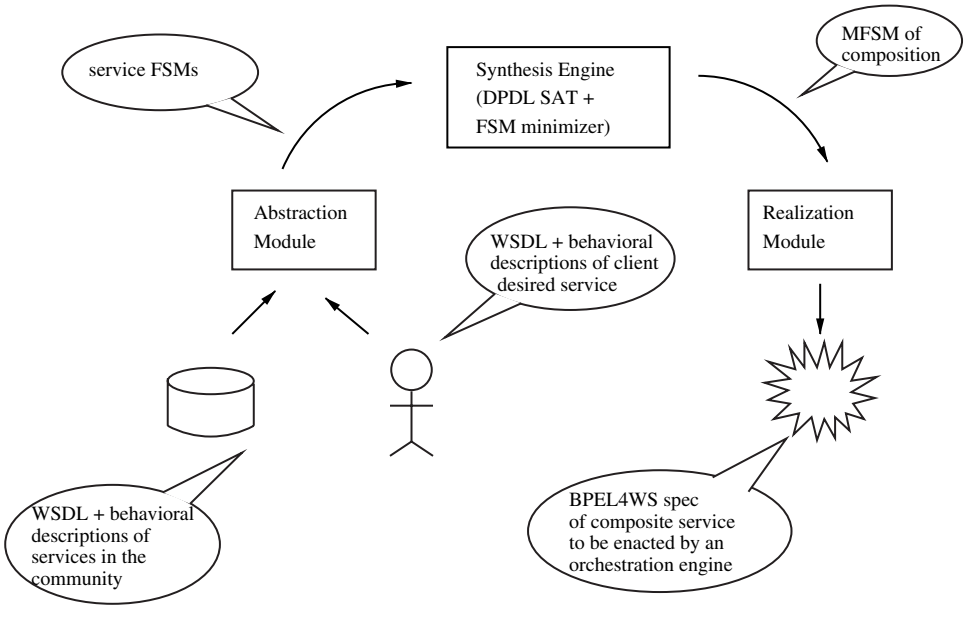


Fig. 10. The service composition architecture.

[i]Note that such behavioral description of services specifies the external schema.

Language,[18] BPEL4WS,[4] 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, Fig. 11 shows the WSDL interface of service E_0 whose behavior is represented in Fig. 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.[56] The client specifies his target service in terms of a WSDL document and of its behavioral description, again expressed using one of the language mentioned before.[j] 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 Φ (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[k] (`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.[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.[18]

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[7]), exploiting the well known correspondence between DPDL formulas and DL knowledge bases.[l] Tableaux algorithms for DLs have been widely studied in the literature, therefore, one can use current highly optimized DL-based systems[34,37] 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[m] (of the DL knowledge base that encodes the

[j]The behavioral description of both the client specification and the services in the repository are expressed in the same language.
[k]It represents the internal schema for the target service.
[l]In fact, current Description Logics systems cannot handle Kleene star. However, since in our DPDL formula Φ, ^{j} is only used to mimic universal assertions, and such systems have the ability to handle universal assertions, they can indeed check satisfiability of Φ.
[m]If one exists.

```
<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>
```

Fig. 11. WSDL specification of service E_0 whose external schema A_0 is represented in Fig. 3(a).

specific composition problem) which is a Mealy composition. For our purpose the well-known $\mathcal{ALC}$,[7] equipped with the ability of expressing axioms, suffices.[15]

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 `Abstraction Module`, and translates them into an $\mathcal{ALC}$ knowledge base (details of the encoding are presented in Ref. 15).
- The $\mathcal{ALC}$ `Tableau Algorithm` module implements the standard tableau algorithm for $\mathcal{ALC}$ (cf. Buchheit *et al.*, 1993[20]). 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*.[52] 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 Refs. 8 and 16):

- 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 Fig. 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.

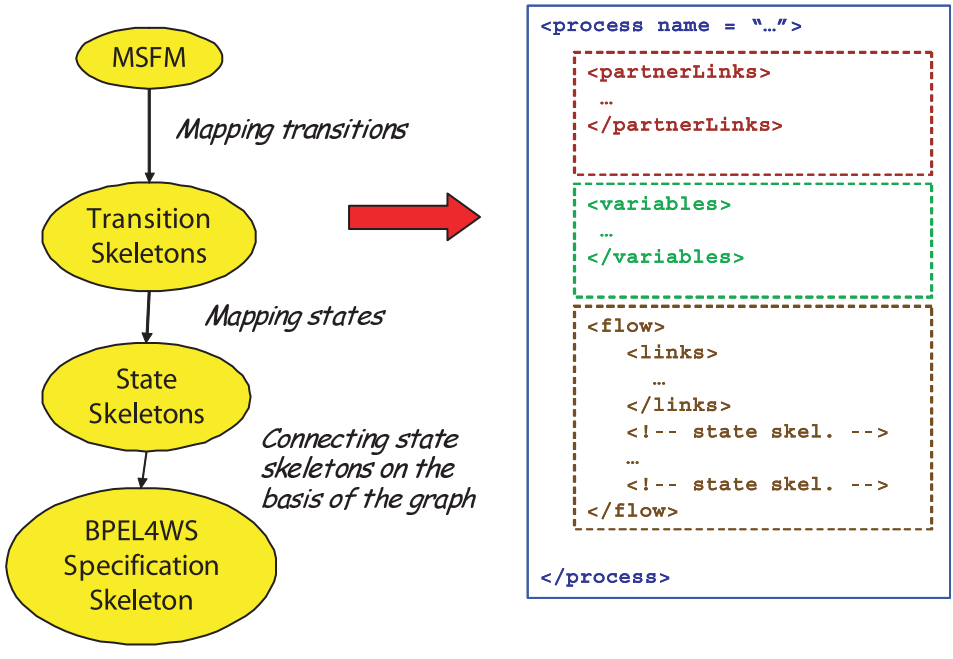


Fig. 12. Methods for deriving the BPEL4WS file and its structure, as inspired by Baïna *et al.*[8]

- 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 `<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 `<pick>` blocks are enclosed in a surrounding `<flow>`; the dependencies are modeled as `<link>`s: `<link>`s are controlled by specific variables `Si-to-Sj` that are set to `TRUE` iff the transition $S_i \rightarrow S_j$ is executed; each state skeleton has many outgoing `<link>`s as states connected in output, each going to the appropriate `<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 `<while>` block controlled by a condition `!exit`, where `exit` is a variable defined *ad hoc* and it is set to `FALSE` by any transition that "goes out" of the cycle; (iii) connect the overall `<while>` block to other state skeletons by appropriate `<link>`s.

```
<onMessage ... >
   <sequence>
      <assign>
         <copy>
            <from variable="input" ... />
            <to variable="transitionData" ... />
         </copy>
      </assign>
      < !-- invocation of the component service -->
      <assign>
         <copy>
            <from variable="transitionData" ... />
            <to variable="output" ... />
         </copy>
      </assign>
      <reply ... />
      </sequence>
</onMessage>
```

(a) Transition skeleton.

```
<!-- N transition from state Si -->
<pick name = "Si">
   <!-- transition #1 -->
      <onMessage ...>
      <!-- transition skeleton -->
      </onMessage>
      ...
      <!-- transition #N -->
      <onMessage ...>
      <!-- transition skeleton -->
      </onMessage>
</pick>
```

(b) State skeleton.

Fig. 13. BPEL4WS code skeletons for transitions and states.

There are some interesting special cases: (i) a state S with self-transitions can be represented as a `<pick>` block enclosed in a `<while>` block controlled by a condition `(Vs)` (the variable `Vs` is set to `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 Fig. 14 the BPEL4WS pseudo-code for the MFSM of the running example.

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,[24] WSCL,[33] WSCI,[6] BPEL4WS,[4] WS-CDL[39]) and

```
<process suppressJoinFailure = "no">

   <partnerLinks>
   ...
   <!-- Sono definiti i partner link per il servizio composto
        (MP3CompositeServiceType), per il servizio componente di
        tipo 1 (MP3ServiceType1) e per il servizio componente di
        tipo 2 (MP3ServiceType2)
   -->
   </partnerLinks>

   <variables>
      <variable name="input" messageType="listen_request"/>
      <variable name="output messageType="listen_response"/>
      <variable name="dataIn" messageType="listen_request"/>
      <variable name="dataOut" messageType="listen_response"/>
      ... ...
   </variables>

   <flow>
      <links>
         <link name="start-to-1"/>
         <link name="start-to-2"/>
      </links>

      <pick createInstance = "yes">
      <!-- A new process instance is created in the initial state.
           This resolve the presence of the cycles without the need
           of an enclosing <while>
      -->

         <onMessage="a" ...>
            <sequence>
               <assign><copy>...</copy></assign>
               ...
               <!-- The "a" transition skeleton should
                    set variables: start-to-1 = TRUE
                                   start-to-2 = FALSE
               -->
               <assign><copy>...</copy></assign>
               <reply ... />
            </sequence>
         </onMessage>
         <onMessage="t" ...>
            <sequence>
               <assign><copy>...</copy></assign>
               ...
               <!-- The "t" transition skeleton should
                    set variables: start-to-1 = FALSE
                                   start-to-2 = TRUE
               -->
               <assign><copy>...</copy></assign>
               <reply ... />
            </sequence>
         </onMessage>
         <source linkName="start-to-1"
          transitionCondition = "bpws:getVariableData(start-to-1) = TRUE" />
         <source linkName="start-to-2"
          transitionCondition = "bpws:getVariableData(start-to-2) = TRUE" />
      </pick>
```

Fig. 14. BPEL4WS pseudo-code for the MFSM shown in Fig. 9. The detail of a transition skeleton is shown only for one operation.

```
      <pick>
         <onMessage partnerLink="client"
                    portType="MP3CompositeServiceType"
                    operation="listen"
                    variable="input">
            <sequence>
               <assign>
                  <copy>
                     <from variable="input" part="selectedSong"/>
                     <to variable="dataIn" part="selectedSong"/>
                  </copy>
               </assign>
               <invoke partnerLink="service"
                       portType="MP3ServiceType1"
                       operation="listen"
                       inputVariable="dataIn"
                       inputVariable="dataOut" />
               <assign>
                  <copy>
                     <from variable="dataOut" part="MP3FileURL"/>
                     <to variable="output" part="MP3FileURL"/>
                  </copy>
               </assign>
               <reply name="replyOutput"
                      partnerLink="client"
                      portType="MP3CompositeServiceType"
                      operation="listen"
                      variable="output"/>
            </sequence>
         </onMessage>
         <target linkName="start-to-1"/>
      </pick>

      <pick>
         <onMessage ... >
            <sequence>
               <assign><copy>...</copy></assign>
               ...
               <assign><copy>...</copy></assign>
               <reply ... />
            </sequence>
         </onMessage>
         <target linkName="start-to-2"/>
      </pick>

   </flow>
</process>
```

Fig. 14. (*Continued*)

protocols (such as SOAP and XML Protocol[58]), 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 an abstract 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[5] defines a specific ontology and a related language for services. A service (i) presents a Service Profile, i.e. what it does, in terms of inputs and outputs, preconditions and effects, (ii) it is described by a Service Process Model, i.e. how it works, in terms of the abstract *internal* process, and (iii) it supports a Service Grounding, i.e. how to access the service, in terms of communication protocol, marshalling and serialization, etc. Services, whose service process model is characterized by a deterministic FSM-based conceptual model could be composed using our technique.

In Bultan *et al.*, 2003,[21] 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, when queues are bounded, they can be represented in FSM by making queues being part of the state. Hence, the two models have the same expressive power.

Service Discovery. Works on service discovery are based on various techniques, such as information retrieval techniques,[59] peer-to-peer scenarios[53] and graph-based techniques in the context of OWL-S services.[13]

In particular, for instance, in van den Heuvel *et al.*, 2001,[57] 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.

We would like to remark that all such approaches take into account only "static" service signatures. We argue that one could use behavioral descriptions as well, and that they could improve the quality of the discovery process.

Service Composition. In Yang and Papazoglou, 2004,[60] 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[46] and in McIlraith *et al.*, 2001,[47] a Situation Calculus-based framework for services is proposed, where a service is seen by the client 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 procedure calls. Therefore, the client cannot control the interleaved execution of "pieces of" services. Another difference is that in McIlraith and Son, 2002[46] and in McIlraith *et al.*, 2001[47] the client specifies his goal to achieve with the composition but 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[47] 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,[21] 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, in our approach the behavior of the services is given, and the synthesis phase reuses them trying to *assemble* them in order to provide the desired behavior. Another difference consists in the nature of the specification of the composition: in Bultan *et al.*, 2003,[21] a linear-time semantics is taken: composition focuses on linear sequences (i.e. paths) of actions. In our approach, the client specification is based on a branching time-semantics: composition focuses on a tree-based structure, where each node denotes a choice point on what to do next.

In Aiello *et al.*, 2002,[1] composition of services is studied based on global goals specifying a complex behavior and expressed in a request language developed for planning under uncertainty; such a goal concerns the client and all the component services.

Pistore *et al.*, 2004,[50] and Traverso and Pistore, 2004,[55] basing on such an approach, present a composition technique that takes as input (i) a set of partially specified services, including one representing the client behavior, modeled as nondeterministic finite state machines, and (ii) a global goal expressed as a branching temporal formula, and returns a plan that specifies how to coordinate the execution of various services in order to realize the global goal. The plan can then be encoded in standard coordination languages and executed by orchestration engines. Note that, differently from our approach, the composition is not tailored towards satisfying completely the client requested behavior, but concerned with the global behavior of the system in which some client desired executions may happen not to be fulfilled.

Lazovik *et al.*, 2003[43] instead 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.

Note that such a form of composition is tightly related to Classical Planning in AI, and has been adopted by many other works.[3] These, although different in the kind of goals, are all based on the idea of sequential composition of the available services, which are considered as black boxes. In contrast, in our proposal the composition involves the concurrent executions of several services, as in Refs. 21, 46, 47, 50, 55. Also our proposal is characterized by the fact 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. We even[21,46,47] 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 Refs. 50 and 55 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.

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.[35,44] 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.[32] The other end of the spectrum is much less studied. Our proposal, together with those in Refs. 21, 46, 47, 50 and 55, tries to explore such an end of the spectrum. Observe that in our setting introducing data in a naive way, i.e. putting the data in the states, is in fact possible. But this would make composition exponential in the data, which is unacceptable, since the amount of data is typically huge (w.r.t. the size of the services) and hence one wants to keep the computations polynomial in it. 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,[30] where verification of BPEL4WS specifications is carried out exploiting model checking techniques, in Narayanan and McIlraith, 2002,[48] where OWL-S services are analyzed exploiting Petri Nets, and in Deutsch *et al.*, 2004,[28] 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* synthesizes a composite service that is correct by construction according to Sec. 4.

Orchestration. Orchestration requires that the composite service is completely specified, in terms of both the specification on how various component services are linked, and the internal process flow of the composite one. In Hull *et al.*, 2003,[38] different technologies, standards and approaches for specification of composite services are considered, including BPEL4WS, BPML, AZTEC, etc. In particular, Hull *et al.*, 2003[38] 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,[22] AZTEC,[25] WISE,[42] MENTOR-lite,[54] e-ADOME[23]): they can be classified into the mediated approach to composition. An interesting case is Self-Serv,[11] 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; the Mealy composition, obtained with our technique, can be translated into

a specific orchestration language, that can then be orchestrated by any orchestration platform, thus obtaining an all system-level guarantees needed in complex distributed applications.

7. Final Remarks and Future Work

The main contribution of this paper w.r.t. 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 services is expressed as finite state machine, and providing a computational complexity characterization of the algorithm for automatic composition (for the lower bound the problem remains open); (iii) presenting $\mathcal{ESC}$, an open source prototype tool that implements our technique for automatically synthesizing a composition.

In Ref. 19, 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 Sec. 6, so that the combination of data and process allows us to devise algorithms for automatic service composition with reasonable computational complexity.

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 nondeterministic 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" subproject 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 his useful comments and discussions; Alessandro Iuliani, for collaborating in the design and realization of the $\mathcal{ESC}$ tool, and Alessia Candido for her technical support with BPEL4WS. Finally,

the authors would like to thank the anonymous reviewers for their valuable suggestions in improving the paper.

References

1. M. Aiello, M. P. Papazoglou, J. Yang, M. Carman, M. Pistore, L. Serafini and P. Traverso, A request language for web-services based on planning and constraint satisfaction, in *Proc. 3rd VLDB Int. Workshop on Technologies for e-Services (VLDB-TES 2002)*, Hong Kong, China (2002).
2. G. Alonso, F. Casati, H. Kuno and V. Machiraju, *Web Services. Concepts, Architectures and Applications* (Springer-Verlag, 2004).
3. J. L. Ambite and J. Blythe, *ICAPS 2004 Workshop on Planning and Scheduling for Web and Grid Services (P4WGS 2004)* (2004), http://www.isi.edu/ikcap/icaps04-workshop/.
4. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic and S. Weerawarana, Business process execution language for web services (Version 1.1) (May 2004), http://www-106.ibm.com/developerworks/library/ws-bpel/.
5. A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne and K. Sycara, DAML-S: Web service description for the semantic web, in *Proc. 1st Int. Semantic Web Conference (ISWC 2002)*, Chia, Sardegna, Italy (2002).
6. A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickovic and S. Zimek, Web Service Choreography Interface (WSCI) 1.0. W3C Note (8 August 2002), http://www.w3.org/TR/wsci/.
7. F. Baader, D. Calvanese, D. McGuinness, D. Nardi and P. F. Patel-Schneider (eds.), *The Description Logic Handbook: Theory, Implementation and Applications* (Cambridge University Press, 2003).
8. K. Baïna, B. Benatallah, F. Casati and F. Toumani, Model-driven web service development, in *Proc. 16th Int. Conf. Advanced Information Systems Engineering (CAiSE 2004)*, Lectures Notes in Computer Science, Vol. 3084 (Springer-Verlag, 2004), pp. 290–300.
9. C. Batini and M. Mecella, Enabling Italian *e*-government through a cooperative architecture, *IEEE Computer* **34**(2) (2001).
10. M. Ben-Ari, J. Y. Halpern and A. Pnueli, Deterministic propositional dynamic logic: Finite models, complexity, and completeness, *J. Computer and System Sciences* **25** (1982) 402–417.
11. B. Benatallah, Q. Z. Sheng and M. Dumas, The self-serv environment for web services composition, *IEEE Internet Computing* **7**(1) (2003).
12. B. Benatallah, F. Casati, F. Toumani and R. Hamadi, Conceptual modeling of web service conversations, in *Proc. 15th Int. Conf. Advanced Information Systems Engineering (CAiSE 2003)* (Springer-Verlag, 2003), pp. 449–467.
13. B. Benatallah, M. S. Hacid, C. Rey and F. Toumani, Request rewriting-based web service discovery, in *Proc. Int. Semantic Web Conf.* (2003).
14. D. Berardi, Automatic service composition. Models, techniques and tools, PhD thesis, Università di Roma "La Sapienza" (2005).
15. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini and M. Mecella, Service composition by description logic based reasoning, in *Proc. Int. Workshop on Description Logics (DL03)*, Rome, Italy (2003).

16. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini and M. Mecella. $\mathcal{ESC}$: A tool for automatic composition of e-services based on logics of programs, in *Proc. 5th VLDB Int. Workshop on Technologies for e-Services (VLDB-TES 2004)* (2004), to appear as Post-Proceedings.
17. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini and M. Mecella, A foundational vision of services, in *Proc. CAiSE 2003 Workshop on Web Services, e-Business, and the Semantic Web (WES 2003)*, Velden, Austria (2003).
18. D. Berardi, F. De Rosa, L. De Santis and M. Mecella, Finite state automata as conceptual model for e-services, *J. Integrated Design and Process Science* (2004), to appear.
19. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini and M. Mecella, Synthesis of underspecified composite e-services based on automated reasoning, in *Proc. 2nd Int. Conf. Service Oriented Computing (ICSOC 2004)* (2004).
20. M. Buchheit, F. M. Donini and A. Schaerf, Decidable reasoning in terminological knowledge representation systems, *J. Artificial Intelligence Research* **1** (1993) 109–138.
21. T. Bultan, X. Fu, R. Hull and J. Su, Conversation specification: A new approach to design and analysis of e-service composition, in *Proc. WWW 2003 Conference*, Budapest, Hungary (2003).
22. F. Casati and M. C. Shan, Dynamic and adaptive composition of *e*-Services, *Information Systems* **6**(3) (2001).
23. D. K. W. Chiu, K. Karlapalem and Q. Li, E-ADOME: A framework for enacting *e*-services, in *Proc. 1st VLDB Int. Workshop on Technologies for e-Services (VLDB-TES 2000)*, Cairo, Egypt (2000).
24. E. Christensen, F. Curbera, G. Meredith and S. Weerawarana, Web Services Description Language (WSDL) 1.1. W3C Note (15 March 2001), http://www.w3.org/TR/wsdl.
25. V. Christophides, R. Hull, G. Karvounarakis, A. Kumar, G. Tong and M. Xiong, Beyond discrete *e*-services: Composing session-oriented services in telecommunications, in *Proc. 2nd VLDB Int. Workshop on Technologies for e-Services (VLDB-TES 2001)*, Rome, Italy (2001).
26. E. Colombo, C. Francalanci, B. Pernici, P. Plebani, M. Mecella, V. De Antonellis and M. Melchiori, Cooperative information systems in virtual districts: The VISPO approach, *IEEE Data Engineering Bulletin* **25**(4) (2002).
27. G. De Giacomo and F. Massacci, Combining deduction and model checking into tableaux and algorithms for converse-PDL, *Information and Computation* **160**(1–2) (2000) 117–137.
28. A. Deutsch, L. Sui and V. Vianu, Specification and verification of data-driven web services, *Symp. Principles of Database Systems (PODS04)* (2004).
29. M. J. Fischer and R. E. Ladner, Propositional dynamic logic of regular programs, *J. Computer and System Sciences* **18** (1979) 194–211.
30. X. Fu, T. Bultan and J. Su, Analysis of interacting BPEL web services, in *Proc. WWW 2004* (May 2004).
31. M. Ghallab, D. Nau and P. Traverso, *Automated Task Planning: Theory & Practice* (Morgan Kaufmann, 2004).
32. S. Ghandeharizadeh, C. A. Knoblock, C. Papadopoulos, C. Shahabi, E. Alwagait, J. L. Ambite, M. Cai, C. Chen, P. Pol, R. R. Schmidt, S. Song, S. Thakkar and R. Zhou, Proteus: A system for dynamically composing and intelligently executing web services, in *Proc. Int. Conf. Web Services (ICWS'03)* (2003).

33. A. Karp, H. Kuno, M. Lemon and D. Beringer, Conversations + interfaces = business logic, in *Proc. 2nd VLDB Int. Workshop on Technologies for e-Services (VLDB-TES 2001)*, Rome, Italy (2001).
34. V. Haarslev and R. Möller, RACER system description, in *Proc. IJCAR 2001*, Lecture Notes in Computer Science, Vol. 2083 (Springer-Verlag 2001), pp. 701–705.
35. A. Y. Halevy, Answering queries using views: A survey, *Very Large Database J*, **10**(4) (2001) 270–294.
36. D. Harel, D. Kozen and J. Tiuryn, *Dynamic Logic* (The MIT Press, 2000).
37. I. Horrocks, The FaCT System, in ed. Harrie de Swart, in *Proc. TABLEAUX'98*, Vol. 1397 (Springer-Verlag, 1998), pp. 307–312.
38. R. Hull, M. Benedikt, V. Christophides and J. Su, E-Services: A look behind the curtain, in *Proc. PODS 2003 Conf.*, San Diego, CA, USA (2003).
39. N. Kavantzas, D. Burdett and G. Ritzinger, Web Services Choreography Description Language (WS-CDL) 1.0. W3C Working Draft, (27 April 2004), http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/.
40. D. Kozen and J. Tiuryn, Logics of programs, in *Handbook of Theoretical Computer Science — Formal Models and Semantics*, ed. J. Van Leeuwen (Elsevier Science Publishers, North-Holland, Amsterdam, 1990).
41. U. Kuter, E. Sirin, D. Nau, B. Parsia and J. Hendler, Information gathering during planning for web service composition, in *Proc. ICAPS-P4WGS 2004* (2004).
42. A. Lazcano, G. Alonso, H. Schuldt and C. Schuler, The WISE approach to electronic commerce, *Int. J. Computer Systems Science & Engineering* **15**(5) (2000).
43. A. Lazovik, M. Aiello and M. P. Papazoglou, Planning and monitoring the execution of web service requests, in *Proc. 1st Int. Conf. Service Oriented Computing (ICSOC 2003)*, Lecture Notes in Computer Science, Vol. 2910 (Springer-Verlag, 2003), pp. 335–350.
44. M. Lenzerini, Data integration: A theoretical perspective, in *Proc. PODS 2002* (2002), pp. 233–246.
45. E. Martinez and Y. Lesperance, Web service composition as a planning task: Experiments using knowledge-based planning, in *Proc. ICAPS-P4WGS 2004* (2004).
46. S. McIlraith and T. Son, Adapting golog for composition of semantic web services, in *Proc. 8th Int. Conf. Knowledge Representation and Reasoning (KR 2002)*, Toulouse, France (2002).
47. S. McIlraith, T. C. Son and H. Zeng, Semantic web services, *IEEE Intelligent Systems* **16**(2) (2001).
48. S. Narayanan and S. McIlraith, Simulation, verification and automated composition of web services, in *Proc. 11th Int. Conf. World Wide Web*, Hawaii, USA (2002).
49. M. P. Papazoglou and D. Georgakopoulos, Service oriented computing (special issue), *Communications of the ACM* **46**(10) (2003).
50. M. Pistore, F. Barbon, P. Bertoli, D. Shaparau and P. Traverso, Planning and monitoring web service composition, *The 11th Int. Conf. Artificial Intelligence, Methodologies, Systems, and Applications (AIMSA04)* (2004). Also presented at the ICAPS'04 Workshop on Planning and Scheduling for Web and Grid Service (P4WGS 2004).
51. R. Reiter, *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems* (The MIT Press, 2001).
52. R. H. Katz, *Contemporany Logic Design* (Benjamin Commings/Addison Wesley Publishing Company, 1993).
53. C. Schmidt and M. Parashar, A peer-to-peer approach to web service discovery, *World Wide Web Journal* **7**(2) (2004).

54. G. Shegalov, M. Gillmann and G. Weikum, XML-enabled workflow management for *e*-Services across heterogeneous platforms, *Very Large Database J.* **10**(1) (2001).
55. P. Traverso and M. Pistore, Automated composition of semantic web services into executable processes, in *Proc. 3rd Int. Semantic Web Conference* (2004), pp. 380–394.
56. UDDI.org. UDDI Technical White Paper (2000) http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf).
57. W. J. van den Heuvel, J. Yang and M. P. Papazoglou, Service representation, discovery and composition for *e*-marketplaces, in *Proc. 9th Int. Conf. Coop. Inf. Syst. (CoopIS 2001)*, Trento, Italy (2001).
58. W3C. XML Protocol, XML Protocol Working Group Web Page, http://www.w3.org/2000/xp/Group/.
59. Y. Wang and E. Stroulia, Flexible interface matching for web-service discovery, in *Proc. 4th Int. Conf. Web Information System Engineering (WISE 2003)* (2003).
60. J. Yang and M. P. Papazoglou, Service components for managing the life-cycle of service compositions, *Information Systems* **29**(2) (2004).
61. J. Yang and M. P. Papazoglou, Web components: A substrate for web service reuse and composition, in *Proc. 14th Int. Conf. Advanced Information Systems Engineering (CAiSE'02)*, Toronto, Canada (2002).