Automatic Composition of *e*-Services ⋆

Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo
Maurizio Lenzerini, and Massimo Mecella

Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italy
lastname@dis.uniroma1.it

Abstract. The main focus of this paper is on automatic *e*-Service composition. We start by developing a framework in which the exported behavior of an *e*-Service is described in terms of its possible executions (execution trees). Then we specialize the framework to the case in which such exported behavior (i.e., the execution tree of the *e*-Service) is represented by a finite state machine. In this specific setting, we analyze the complexity of synthesizing a composition, and develop sound and complete algorithms to check the existence of a composition and to return one such a composition if one exists. To the best of our knowledge, our work is the first attempt to provide an algorithm for the automatic synthesis of *e*-Service composition, that is both proved to be correct, and has an associated computational complexity characterization.

1 Introduction

Service Oriented Computing (SOC [20]) aims at building agile networks of collaborating business applications, distributed within and across organizational boundaries.[1] *e*-Services, which are the basic building blocks of SOC, represent a new model in the utilization of the network, in which self-contained, modular applications can be described, published, located and dynamically invoked, in a programming language independent way.

The commonly accepted and *minimal* framework for *e*-Services, referred to as Service Oriented Architecture (SOA [21]), consists of the following basic roles: *(i)* the *service provider*, which is the subject (e.g., an organization) providing services; *(ii)* the *service directory*, which is the subject providing a repository/registry of service descriptions, where providers publish their services and requestors find services; and, *(iii)* the *service requestor*, also referred to as client, which is the subject looking for and invoking the service in order to fulfill some goals. A requestor discovers a suitable service in the directory, and then it connects to the specific service provider and uses the service.

Research on *e*-Services spans over many interesting issues regarding, in particular, composability, synchronization, coordination, and verification [26]. In this paper, we are particularly interested in automatic *e*-Service composition. *e*-Service *composition* addresses the situation when a client request cannot be satisfied by an available *e*-Service, but a *composite* *e*-Service, obtained by combining "parts of" available *component* *e*-Services, might be used. Each composite *e*-Service can be regarded as a kind of client wrt its components, since it (indirectly) looks for and invokes them. *e*-Service composition leads to enhancements of the SOA, by adding new elements and roles, such as brokers and integration systems, which are able to satisfy client needs by combining available *e*-Services.

Composition involves two different issues. The first, sometimes called *composition synthesis*, or simply *composition*, is concerned with synthesizing a new composite *e*-Service, thus producing a specification of how to coordinate the component *e*-Services to obtain the composite *e*-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, often referred to as *orchestration*, is concerned with coordinating the various component *e*-Services according to some given specification, and also monitoring control and data flow among the

⋆ This work has been partially supported by MIUR through the "Fondo Strategico 2000" Project *VISPO* and the "FIRB 2001" Project *MAIS*. The work of Massimo Mecella has been also partially supported by the European Commission under Contract No. IST-2001-35217, Project *EU-PUBLI.com*.

[1] cf., Service Oriented Computing Net: `http://www.eusoc.net/`

involved e-Services, in order to guarantee the correct execution of the composite e-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 e-Services. Note that several works published in the literature address service oriented computing from different points of views (see [13] for a survey), but an agreed comprehension of what an e-Service is, in an abstract and general fashion, is still lacking. Our framework, although simplified in several aspects, provides not only a clear definition of e-Services, but also a formal setting for a precise characterization of automatic composition of e-Services.

The second contribution of the paper is an effective technique for automatic e-Service composition. In particular, we specialize the general framework to the case where e-Services are specified by means of finite state machines, and we present an algorithm that, given a specification of a target e-Service, i.e., specified by a client, and a set of available e-Services, synthesizes a composite e-Service that uses only the available e-Services and fully captures the target one. We also study the computational complexity of our algorithm, and we show that it runs in exponential time with respect to the size of the input state machines.

Although several papers have been already published that discuss either a formal model of e-Services (even more expressive than ours, see e.g., [7]), or propose algorithms for computing composition (e.g., [19]), to the best of our knowledge, the work presented in this paper is the first one tackling simultaneously the following issues: *(i)* presenting a formal model where the problem of e-Service composition is precisely characterized, *(ii)* providing techniques for computing e-Service composition in the case of e-Services represented by finite state machines, and *(iii)* providing a computational complexity characterization of the algorithm for automatic composition.

The rest of this paper is organized as follows. In Section 2 and 3 we define our general formal framework, and in Section 4 we define the problem of composition synthesis in such a framework. In Section 5 we specialize the general framework to the case where e-Services are specified by means of finite state machines, and in Section 6 we present an EXPTIME algorithm for automatic e-Service composition in the specialized framework. Finally, in Section 7 we consider related research work and in Section 8 we draw conclusions by discussing future work.

2 General Framework

Generally speaking, an e-Service is a software artifact (delivered over the Internet) that interacts with its clients in order to perform a specified task. A client can be either a human user, or another e-Service. When executed, an e-Service performs its task by directly executing certain actions, and interacting with other e-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 an e-Service during its life time, as shown in Figure 1:

- The e-Service *schema* specifies the features of an e-Service, in terms of functional and non-functional requirements. Functional requirements represent *what* an e-Service does. All other characteristics of e-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 use the term "e-Service schema" to denote the specification of functional requirements only.
- The e-Service *implementation and deployment* indicate *how* an e-Service is realized, in terms of software applications corresponding to the e-Service schema, deployed on specific platforms. This aspect regards the technology underlying the e-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 e-Services.
- An e-Service *instance* is an occurrence of an e-Service effectively running and interacting with a client. In general, several running instances corresponding to the same e-Service schema exist, each one executing independently from the others.

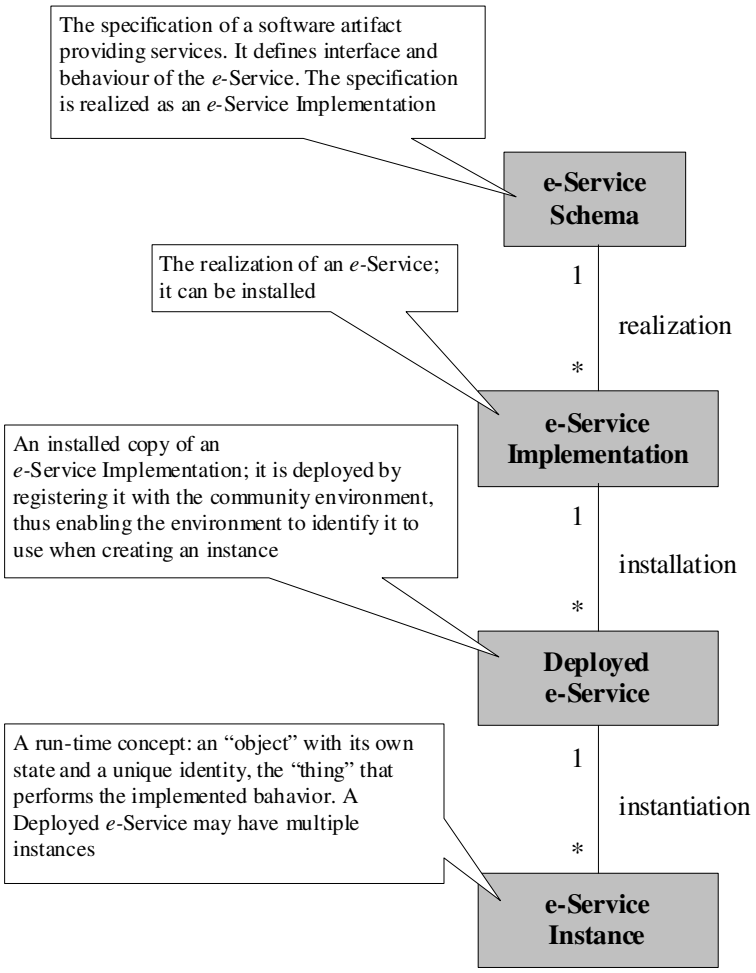


Fig. 1. *e*-Service facets

In order to execute an *e*-Service, the client needs to *activate* an instance from a deployed *e*-Service. In our abstract model, the client can then interact with the *e*-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 information returned the client chooses the next action to invoke. In turn, the activated *e*-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 *e*-Service instance. However, in principle, a given *e*-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 *e*-Service instance is ever executed.

In general, when a client invokes an *e*-Service instance 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 *e*-Service instances. All this is transparent to the client. To precisely capture the situations when the execution of certain actions can be delegated to other *e*-Service instances, we introduce the notion of *community* of *e*-Services, which 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 *e*-Services specified in terms of the common set of actions.

Hence, to join a community, an *e*-Service needs to export its service(s) in terms of the alphabet of the community. The added value of a community is the fact that an *e*-Service of the community may delegate the execution of some or all of its actions to other instances of *e*-Services in the community. We call such an *e*-Service *composite*. If this is not the case, an *e*-Service is called *simple*. Simple *e*-Services realize offered actions directly in the software artifacts implementing them, whereas composite *e*-Services, when receiving requests from clients, can invoke other *e*-Service instances in order to fulfill the client's needs.

Notably, the community can be used to generate (virtual) *e*-Services whose execution completely delegates actions to other members of the community. In other words, the community can be used to realize a target *e*-Service requested by the client, not simply by selecting a member of the community to which delegate the target *e*-Service actions, but more generally by suitably "composing" parts of *e*-Service instances in the community in order to obtain a virtual *e*-Service which is coherent with the target one. This function of composing existing *e*-Services on the basis of a target *e*-Service is known as *e*-Service composition, and is the main subject of the research reported in this paper.

3 *e*-Service Schema

From the external point of view, i.e., that of a client, an *e*-Service E, belonging to a community C, is seen as a black box that exhibits a certain *exported behavior* represented as sequences 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 *e*-Service belonging to the community C with which E interacts, transparently to the client of E. To capture these two points of view we introduce the notion of *e*-Service schema, as constituted by two different parts, called *external schema* and *internal schema*, respectively.

Also *e*-Service instances can be characterized by an external and an internal view: further details can be found in [5].

3.1 External Schema

The aim of the external schema is to specify the exported behavior of the *e*-Service. For now we are not concerned with 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 all possible instances of an *e*-Service. Therefore, any instance of an *e*-Service executes a path of such a tree. In this sense, each node x of

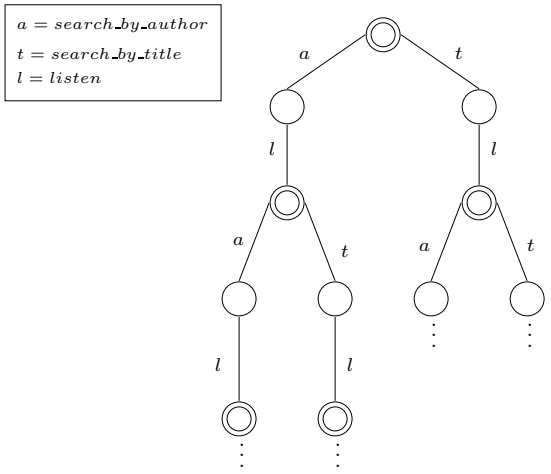


Fig. 2. External execution tree of e-Service E_0

an external execution tree represents the history of the sequence of actions of all e-Service instances[2], that have 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 e-Service should perform. We call the pair $(x, x{\cdot}a)$ *edge* of the tree and we say that such an edge is *labeled* with action a. The root ε of the tree represents the fact that the e-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 e-Service. In other words, the execution of an e-Service can correctly terminate only at these points[3].

Notably, an execution tree does not represent the information returned to the client by the e-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 an e-Service E, we denote with $T(E^{ext})$ the external execution tree *specified* by E^{ext}.

Example 1. Figure 2 shows (a portion of) an (infinite) external execution tree representing e-Service E_0 that allows for searching and listening to `mp3` files[4]. In particular, the client may choose to search for a song by specifying either its author(s) or its title (action `search_by_author` and `search_by_title`, respectively). Then the client selects and listens to a song (action `listen`). Finally, the client chooses whether to perform those actions again. □

3.2 Internal Schema

The internal schema specifies, besides the external behavior of the e-Service, the information on which e-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 e-Service instances executing a. Every element of I is a pair (E', e'), where E' is an e-Service and e' is the identifier of an instance of E'. The identifier e' uniquely identifies the instance of E' within the internal execution tree. In

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

[3] Typically, in an e-Service, the root is final, to model that the computation of the e-Service may not be started at all by the client.

[4] Final nodes are represented by two concentric circles.

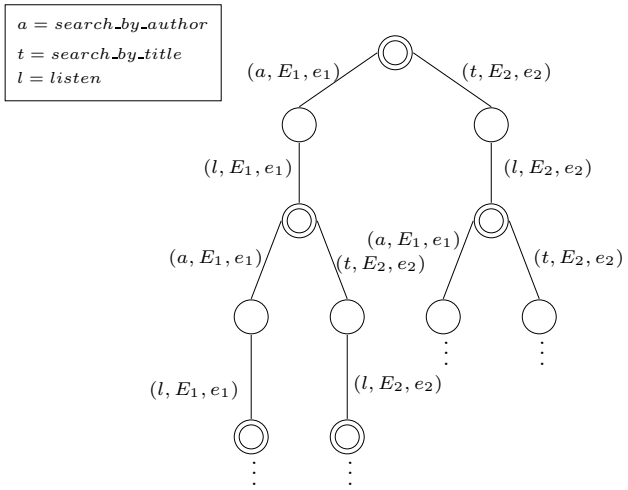


Fig. 3. Internal execution tree of e-Service E_0

general, in the internal execution tree of an e-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 e-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 e-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 an e-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 $T(E^{int})$.

An e-Service E with external schema E^{ext} and internal schema E^{int} is *well formed*, if $T(E^{int})$ conforms to $T(E^{ext})$, i.e., its internal execution tree conforms with its external execution tree.

We now formally define when an e-Service of a community correctly delegates actions to other e-Services of the community. We need a preliminary definition: given the internal execution tree T_{int} of an e-Service E, and a path p in T_{int} starting from the root, we call the *projection* of p on an instance e' of an e-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.

We say that the internal execution tree T_{int} of an e-Service E is *coherent* with a community C if:

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

Observe that, if an e-Service of a community C is simple, i.e., it does not delegate actions to other e-Service instances, then it is trivially coherent with C. Otherwise, it is composite and hence delegates actions to other e-Service instances. In the latter case, the behavior of each one of such e-Service instances must be correct according to its external schema.

A community of e-Services is *well-formed* if each e-Service in the community is *well-formed*, and the internal execution tree of each e-Service in the community is coherent with the community.

Example 2. Figure 3[5] shows (a portion of) an (infinite) internal execution tree, conforming to the external execution tree of e-Service E_0 shown in Figure 2, where all the actions are delegated to e-Services of the

[5] In the figure, each action is delegated to exactly one instance of an e-Service schema. Hence, for simplicity, we have denoted a label $(a, \{(E_i, e_i)\})$ simply by (a, E_i, e_i), for $i = 1, 2$.

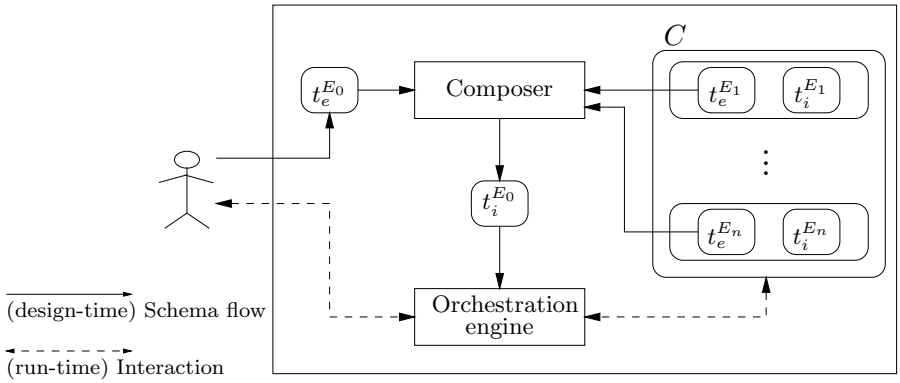


Fig. 4. *e*-Service Integration System

community. In particular, the execution of `search_by_title` action and its subsequent `listen` action are delegated to instance e_2 of *e*-Service E_2, and `search_by_author` action and its subsequent `listen` action to instance e_1 of *e*-Service E_1. □

4 Composition Synthesis

When a user requests a certain service from an *e*-Service community, there may be no *e*-Service in the community that can deliver it directly. However, it may still be possible to synthesize a new composite *e*-Service, which suitably delegates action execution to the *e*-Services of the community, and when suitably orchestrated, provides the user with the service he requested. Formally, given an *e*-Service community C and the external schema E^{ext} of a target *e*-Service E expressed in terms of the alphabet Σ of C, a *composition* of E wrt C is an internal schema E^{int} such that *(i)* $T(E^{int})$ conforms to $T(E^{ext})$, *(ii)* $T(E^{int})$ delegates all actions to the *e*-Services of C (i.e., `this` does not appear in $T(E^{int})$), and *(iii)* $T(E^{int})$ is coherent with C.

The problem of *composition existence* is the problem of checking whether there exists some internal schema E^{int} that is a composition of E wrt C. Observe that, since for now we are not placing any restriction of the form of E^{int}, this corresponds to checking if there exists an internal execution tree T_{int} such that *(i)* T_{int} conforms to $T(E^{ext})$, *(ii)* T_{int} delegates all actions to the *e*-Services of C, and *(iii)* T_{int} is coherent with C.

The problem of *composition synthesis* is the problem of synthesizing an internal schema E^{int} for E that is a composition of E wrt C.

Figure 4 shows the architecture of an *e-Service Integration System*, which delivers possibly composite *e*-Services on the basis of user requests, exploiting the available *e*-Services of a community C. When a client requests a new *e*-Service E, he presents his request in the form of an external *e*-Service schema E^{ext} for E, and expects the *e*-Service Integration System to execute an instance of E. To do so, first a *composer* module makes the composite *e*-Service E available for execution, by synthesizing an internal schema E^{int}[6] of E that is a composition of E wrt the community C. Then, following the internal execution tree $T(E^{int})$ specified by E^{int}, an *orchestration engine* activates an (internal) instance of E, and orchestrates the different available *e*-Services, by activating and interacting with their external view, so as to fulfill the client's needs.

The orchestration engine is also in charge of terminating the execution of component *e*-Service instances, offering the correct set of actions to the client, as defined by the external execution tree, and invoking the action chosen by the client on the *e*-Service that offers it.

All this happens in a transparent manner for the client, who interacts only with the *e*-Service Integration System and is not aware that a composite *e*-Service is being executed instead of a simple one.

[6] If at least one exists.

5 *e*-Services as Finite State Machines

Till now, we have not referred to any specific form of *e*-Service schemas. In what follows, we consider *e*-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).

The class of *e*-Services that can be captured by FSMs are of particular interest. This class allows us to address an interesting set of *e*-Services, that are able to carry on rather complex interactions with their clients, performing useful tasks. Indeed, several papers in the *e*-Service literature adopt FSMs as the basic model of exported behavior of *e*-Services [7, 6]. 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 an *e*-Service in the community that can be involved in the internal execution tree of another *e*-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 *e*-Services in a community is finite, this implies that the overall number of instances orchestrated by the orchestrator in executing an *e*-Service is finite and bounded by the number of *e*-Services belonging to the community. Within this setting, in the next section, we show how to solve the composition problem, 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.

We consider here *e*-Services whose external schemas can be represented with a finite number of states. Intuitively, this means that we can factorize the sequence of actions executed at a certain point into a finite number of states, which are sufficient to determine the future behavior of the *e*-Service. Formally, for an *e*-Service E, the external schema of E is a FSM $A_E^{ext} = (\Sigma, S_E, s_E^0, \delta_E, F_E)$, where:

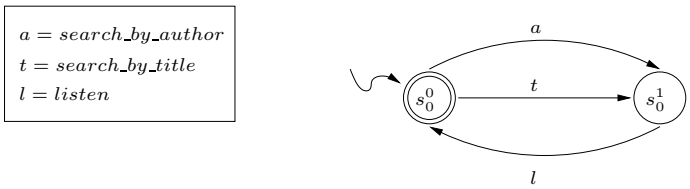
- Σ is the alphabet of the FSM, which is the alphabet of the community;
- S_E is the set of states of the FSM, representing the finite set of states of the *e*-Service E;
- s_E^0 is the initial state of the FSM, representing the initial state of the *e*-Service;
- $\delta_E : S_E \times \Sigma \rightarrow S_E$ 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 *e*-Service E, i.e., the states where the interactions with E can be terminated.

Example 3. Figure 5(a) shows the external schema of the (target) *e*-Service E_0 of Examples 1 and 2, specified by the client as a FSM A_0. Figure 5 (b) and (c) show the external schema, represented as FSMs A_1 and A_2, respectively associated to component *e*-Services E_1 and E_2 of Example 2. In other words, A_1 and A_2 are the external schema of the *e*-Services that should be composed in order to obtain a new *e*-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`).

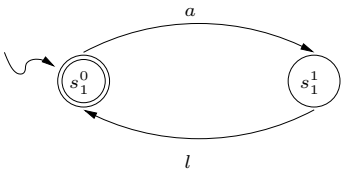
E_1 and E_2 belong to the same community of *e*-Services C. Wlog, we assume that C is composed by only E_1 and E_2 and therefore, the (finite) alphabet of actions of C is $\Sigma = \{\texttt{search_by_author}, \texttt{search_by_title}, \texttt{listen}\}$. According to our setting, the client specifies the external schema A_0 of his target *e*-Service in terms of Σ. □

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

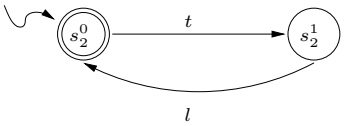
- ε, as usual, is the root of $T(A_E^{ext})$ and $\sigma(\varepsilon) = s_E^0$;
- if x is a node of $T(A_E^{ext})$, and $\sigma(x) = s$, for some $s \in S_E$, then for each a such that $s' = \delta_E(s, a)$ is defined, $x \cdot a$ is a node of $T(A_E^{ext})$ and $\sigma(x \cdot a) = s'$;
- x is final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



(a) External schema A_0 of target e-Service E_0



(b) External schema A_1 of component e-Service E_1



(c) External schema A_2 of component e-Service E_2

Fig. 5. Composition of e-Services

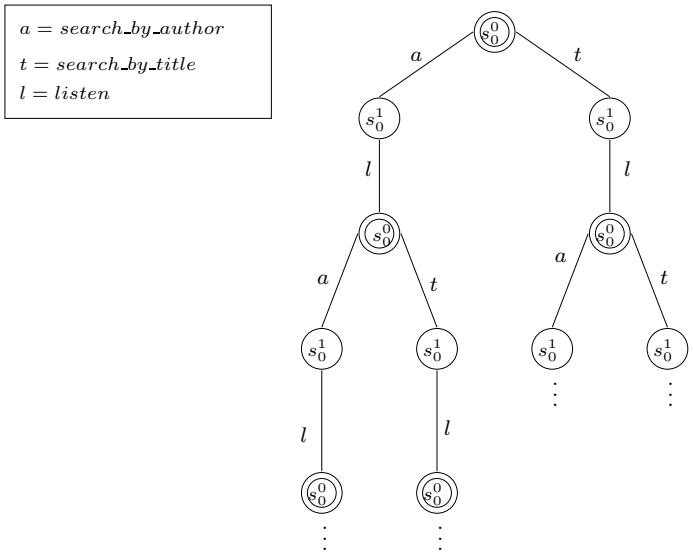


Fig. 6. External execution tree $T(A_0)$.

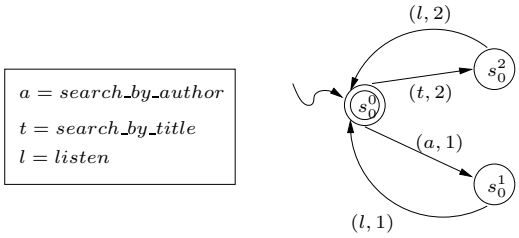


Fig. 7. *e*-Service internal specification as MFSM M_0.

Example 4. Figure 6 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.

$\sigma(\varepsilon) = s_0^0$
$\sigma(a) = \sigma(t) = s_0^1$
$\sigma(a{\cdot}l) = \sigma(t{\cdot}l) = s_0^0$
$\sigma(a{\cdot}l{\cdot}a) = \sigma(a{\cdot}l{\cdot}t) = \sigma(t{\cdot}l{\cdot}a) = \sigma(t{\cdot}l{\cdot}t) = s_0^1$
$\sigma(a{\cdot}l{\cdot}a{\cdot}l) = \sigma(a{\cdot}l{\cdot}t{\cdot}l) = \sigma(t{\cdot}l{\cdot}a{\cdot}l) = \sigma(t{\cdot}l{\cdot}t{\cdot}l) = s_0^0$
...

σ maps over s_0^1 the nodes of the tree that represent strings ending by a or t; it maps over s_0^0 the root and the nodes of the tree associated to strings ending by l. Note that $T(A_0)$ is equal to the external execution tree T_{ext} of Figure 2. 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 FSMs that specify the same execution tree. □

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

We are interested in internal schemas, for an *e*-Service E, that have a finite number of states, i.e., that can be represented as a Mealy FSM (MFSM) $A_E^{int} = (\Sigma, 2^{[n]}, S_E^{int}, s_E^{0\ int}, \delta_E^{int}, \omega_E^{int}, F_E^{int})$, where:

- $\Sigma, S_E^{int}, s_E^{0\ int}, \delta_E^{int}, F_E^{int}$, have the same meaning as for A_E^{ext};
- $2^{[n]}$ is the output alphabet of the MFSM, which is used to denote which *e*-Service instances execute each action;
- $\omega_E^{int} : S_E^{int} \times \Sigma \to 2^{[n]}$ is the output function of the MFSM, that, given a state s and an action a, returns the subset of *e*-Services that executes action a when *e*-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.

Example 5. Figure 7 shows a possible internal schema for the target *e*-Service E_0. It is represented as a MFSM M_0. In the figure, we have defined the output function ω^{int} as follows:

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

□

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

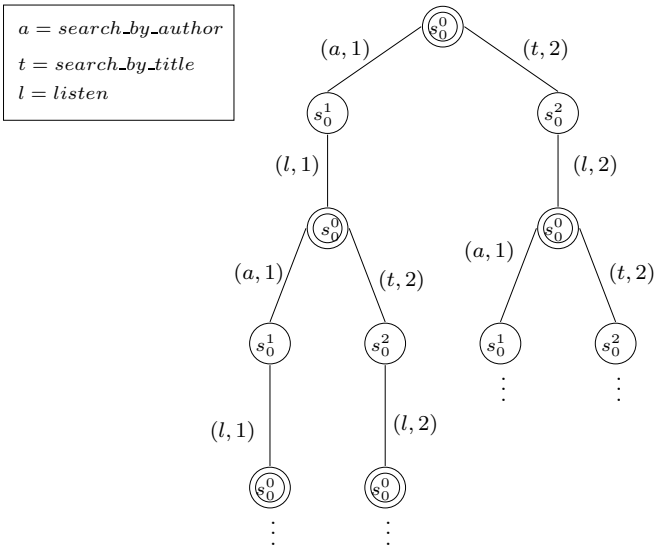


Fig. 8. Internal execution tree $T(M_0)$.

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

Example 6. Figure 8 shows a portion of the internal execution tree $T(M_0)$ defined from M_0, shown in Figure 7. Each node of the tree is labeled with the state of M_0 that mapping σ^{int} from nodes of $T(M_0)$ to states of M_0, associates to it. The mapping σ^{int} is defined as follows.

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

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 Figure 3. 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 Figure 5(a).

Finally, note that in general, a FSM and its corresponding MFSM may have different structures. □

Given an e-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 e-Services whose external schemas are FSMs. In this paper, we do not go into the details of these problems, and instead we concentrate on composition.

6 Automatic *e*-Service Composition

We address the problem of actually checking the existence of a composite e-Service in the FSM-based framework introduced above. We show that if a composition exists then there is one where the internal schema is constituted by a MFSM, and we show how to actually synthesize such a MFSM. The basic tool we use to show such results is reducing the problem of composition existence 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 [15]. We refer to Appendix A for a brief tutorial on DPDL. In order to make clearer the technique, we show how to build a MFSM for the target e-Service whose external schema is represented in Figure 5(a), and for the community of Example 3.

Given the target e-Service E_0 whose external schema is a FSM A_0 and a community of e-Services formed by n component e-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) automaton 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 7. The set $\mathcal{P}$ of atomic propositions is defined as follows:

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

The meaning of atomic propositions is as follows:

- s_j^i, for $i = 0, 1$ and $j = 0, \ldots, 2$: automaton A_j is in state s_j^i
- F_j for $j = 0, \ldots, 2$: automaton A_j is in a final state
- $moved_j$ $j = 1, \ldots, 2$: (component) automaton A_j performed a transition.

The set $\mathcal{A}$ of deterministic atomic actions is defined as follows:

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

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 $[u]$. 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.

The formula Φ is built as a conjunction of the following formulas.

- The 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).
  - $[u](s \rightarrow \langle a \rangle \texttt{true} \wedge [a]s')$ 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 8. Formulas capturing the external schema A_0 of the target e-Service E_0.

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

This formula states that automaton 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 automaton 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.

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

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 automaton 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. □

– For each component FSM $A_i = (\Sigma, S_i, s_i^0, \delta_i, F_i)$, the following formulas:

- $[u](s \rightarrow \neg s')$ 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.
- $[u](s \rightarrow [a](moved_i \wedge s' \vee \neg moved_i \wedge s))$ for each a such that $s' = \delta_i(s, a)$; these encode the transitions of A_i, conditionalized to the fact that the component A_i is actually required to make a transition a in the composition.
- $[u](s \rightarrow [a]\neg moved_i)$ 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 in the composition.
- $[u](F_i \leftrightarrow \bigvee_{s \in F_i} s)$; this highlights final states of A_i.

Example 9. Formulas capturing the external schema A_1 of component e-Service E_1.

$$[u]s_1^0 \rightarrow \neg s_1^1$$

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 automaton 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 automaton, hence A_1 did not move and remained in the current state s_1^0.

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

These formulas encode the situation when a transition is not defined. For example, the first formula states that if the automaton is in state s_1^0 and it receives actions l or t in input, it does not move; this

holds for all possible (previous) sequences of actions. Note that the situation when the automaton 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 automaton A_1.

Formulas capturing the external schema A_2 of component e-Service E_2.

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

$$[u]s_2^0 \rightarrow \neg s_2^1$$

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

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

$$[u](F_2 \leftrightarrow s_2^0)$$

□

– Finally, the following formulas:

- $s_0^0 \wedge \bigwedge_{i=1,\ldots,n} s_i^0$; this says that initially all e-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 e-Service is in a final state also all component e-Services must be in a final state.

Example 10. The following formulas must hold for the overall composition.

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

It asserts that all e-Services start from their initial states.

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

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 e-Service, either E_1 or E_2.
Finally, if the composite e-Service is in a final state, both component e-Services must be in a final state: the composite e-Service may terminate only if also all the component e-Services can.

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

□

It is easy to prove that the Kripke structure for DPDL formula Φ is deterministic, as it should be. Non determinism may be introduced by the operator $\langle\rangle$. However, we are guaranteed that no atomic action a relates state s_1 with two different target states s_1 and s_2, 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 characterized by the same atomic proposition `true`.

Theorem 1. *The DPDL formula Φ, constructed as above, is satisfiable if and only if there exists a composition of E_0 wrt $E_1, \ldots, E_n$.*

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

Then for the target e-Service E_0 and each component e-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 a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \{a^{\mathcal{I}}\}_{a \in \Sigma}, \{P^{\mathcal{I}}\}_{P \in \mathcal{P}})$ of Φ as follows:

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

It is easy to check that, being T_{int} specified by a composition E_{int}, the above model indeed satisfies Φ.

"$\Rightarrow$" Let Φ be satisfiable and $\mathcal{I} = (\Delta^{\mathcal{I}}, \{a^{\mathcal{I}}\}_{a \in \Sigma}, \{P^{\mathcal{I}}\}_{P \in \mathcal{P}})$ be a tree-like 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 possible to show that: *(i)* T_{int} conforms to $T(A_0)$, *(ii)* T_{int} delegates all actions to the e-Services of $E_1, \ldots, E_n$, and *(iii)* T_{int} is coherent with $E_1, \ldots, E_n$. Since we are not placing any restriction on the kind of specification allowed for internal schemas, it follows that there exists an internal schema E_{int} that is a composition of E_0 wrt $E_1, \ldots, E_n$. □

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 1 we get the following complexity result.

Theorem 2. *Checking the existence of an* e*-Service composition can be done in EXPTIME.*

Observe that, because of the small model property, from 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 wrt $E_1, \ldots, E_n$, which has the form of a MFSM. Specifically, given a finite model $\mathcal{I} = (\Delta^{\mathcal{I}}, \{a^{\mathcal{I}}\}_{a \in \Sigma}, \{P^{\mathcal{I}}\}_{P \in \mathcal{P}})$, we define such an MFSM $A_c = (\Sigma, 2^{[n]}, S_c, s_c^0, \delta_c, \omega_c, F_c,)$ as follows:

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

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

Theorem 3. *If there exists a composition of E_0 wrt $E_1, \ldots, E_0$, then there exists one which is a MFSM of at most exponential size in the size of the external schemas A_0, $A_1, \ldots, A_n$ of E_0, $E_1, \ldots, E_n$ respectively.*

Proof (sketch). By Theorem 1, 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 $\mathcal{I}$ of size at most exponential in Φ, and hence in A_0 and $A_1, \ldots, A_n$. From $\mathcal{I}$ we can construct a MFSM A_c as above. It is possible to show that 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 e-Services of $E_1, \ldots, E_n$, and *(iii)* $T(A_c)$ is coherent with $E_1, \ldots, E_n$. □

From a practical point of view, because of the correspondence between Propositional Dynamic Logics (which DPDL belongs to) and Description Logics [8], one can use current highly optimized Description Logic systems [3][7] to check the existence of e-Service compositions. Indeed, these systems are based on tableaux techniques that construct a model when checking for satisfiability, and from such a model one can construct a MFSM that is the composition.

6.1 Building composition

In this subsection, we first show how to build a possibly infinite model $\mathcal{I}$ for the DPDL formula Φ constituted as in the previous section. We follow the proof of Theorem 1 ("$\Leftarrow$" direction). In order to build an internal execution tree for E_0 from FMS A_1 and A_2, i.e., to synthesize a composite e-Service E_0 with components E_1 and E_2 ("$\Rightarrow$" direction), it suffices to repeat the steps backwards. Some of these steps have been discussed in previous examples, but we report them here for sake of readability.

Then, assuming to have derived from $\mathcal{I}$ a finite model $\mathcal{I}_f$ for Φ[8], we show how to devise an internal schema conforming to A_0 that has a finite state representation, and such that all conditions in Section 4 holds.

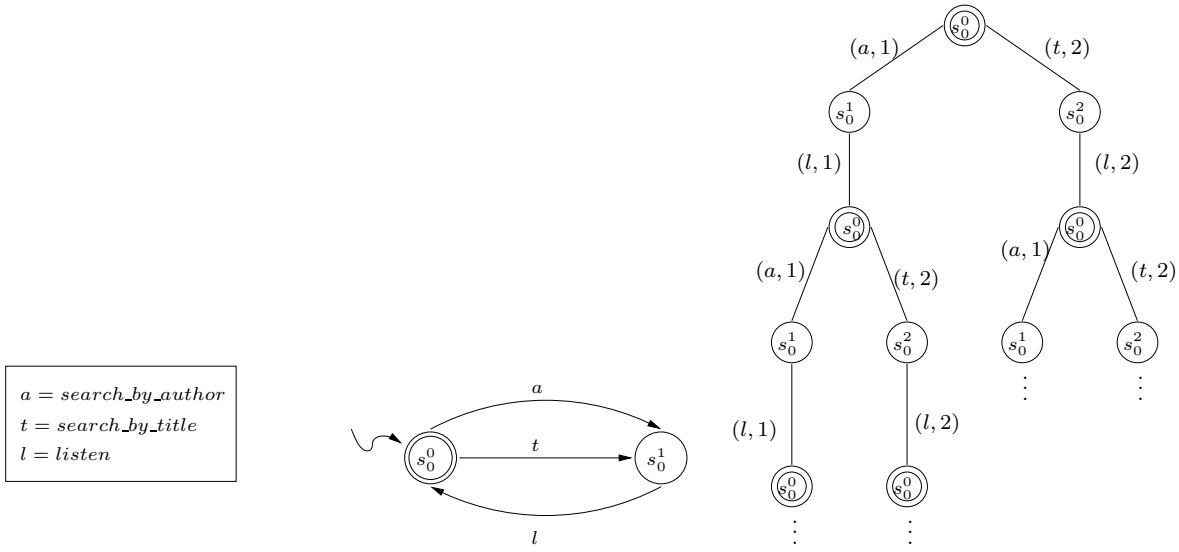
We assume that, given the component FSM A_1 and A_2 there exists a composite e-Service having FSM A_0 as external schema and A_1 and A_2 as components. Let $T({E_0}^{int})$ be the internal execution tree for E_0 wrt the community C to which E_1 and E_2 belong, such that: *(i)* $T({E_0}^{int})$ conforms to $T(A_0)$, i.e., to the external execution tree obtained by A_0 as in Section 5, *(ii)* $T({E_0}^{int})$ delegates all actions to the e-Services of C and in particular to E_1 and E_2, and *(iii)* $T({E_0}^{int})$ is coherent with C.

The mapping σ from nodes of $T({E_0}^{int})$ to states of the automata, is defined as follows by induction on the level of nodes in the tree. The existence of the mapping guarantees that condition *(i)* above is satisfied.

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

[7] In fact, current Description Logics systems cannot handle Kleene star. However, since in Φ, $*$ is only used to mimic universal assertions, and such systems have the ability of handling universal assertions, they can indeed check satisfiability of Φ.

[8] Because of the small model property, we know that this is always possible.



(a) External schema A_0 of target e-Service E_0 (b) The mapping σ on $T({E_0}^{int})$

Fig. 9. Composition of e-Services

Figure 9(b) represents the internal execution tree of E_0, where each node is labeled with the corresponding state of the automaton. σ maps over s_0^1 the nodes of the tree that represent strings ending by a or t; it maps over s_0^0 the root of the tree and the nodes of the tree associated to strings ending by l.

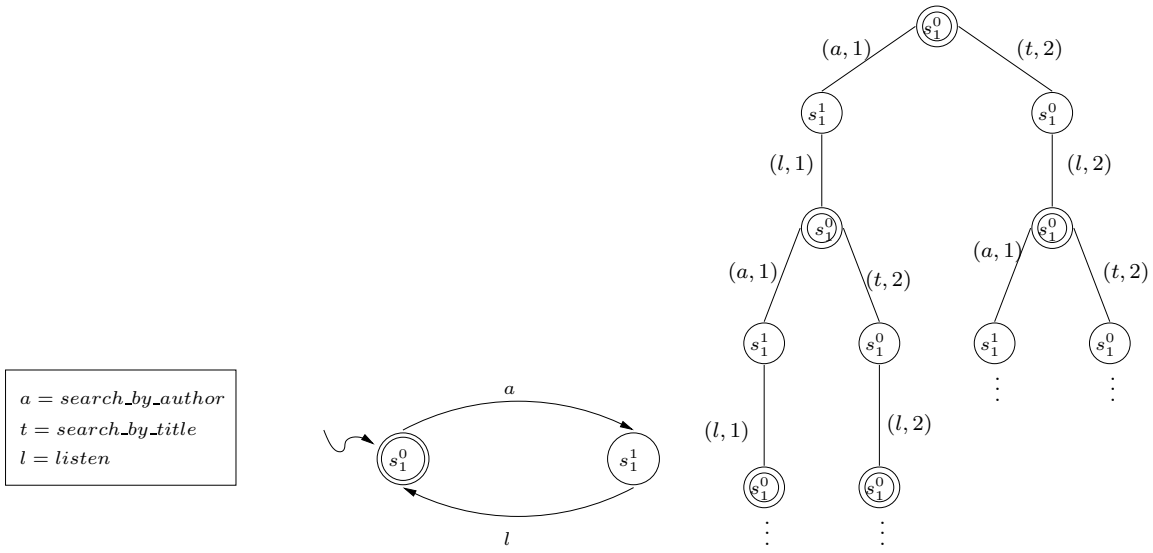
The mapping σ_1 from nodes of $T({E_0}^{int})$ to states of A_1 is defined as follows.

$\sigma_1(\varepsilon) = s_1^0$
$\sigma_1(a) = s_1^1$
$\sigma_1(t) = s_1^0$
$\sigma_1(a{\cdot}l) = \sigma_1(t{\cdot}l) = s_1^0$
$\sigma_1(a{\cdot}l{\cdot}a) = \sigma_1(t{\cdot}l{\cdot}a) = s_1^1$
$\sigma_1(a{\cdot}l{\cdot}t) = \sigma_1(t{\cdot}l{\cdot}t) = s_1^0$
$\sigma_1(a{\cdot}l{\cdot}a{\cdot}l) = \sigma_1(a{\cdot}l{\cdot}t{\cdot}l) = \sigma_1(t{\cdot}l{\cdot}a{\cdot}l) = \sigma_1(t{\cdot}l{\cdot}t{\cdot}l) = s_1^0$
...

Figure 10(b) represents the internal execution tree of E_0, where each node is labeled with the corresponding state of the automaton. σ_1 maps over s_1^1 the nodes of the tree that represent strings ending by a; it maps over s_1^0 the root of the tree and the nodes of the tree associated to strings ending by l or by t. Note that since the automaton is not defined over t, it does not move when it receives t or $t{\cdot}l$ as input.

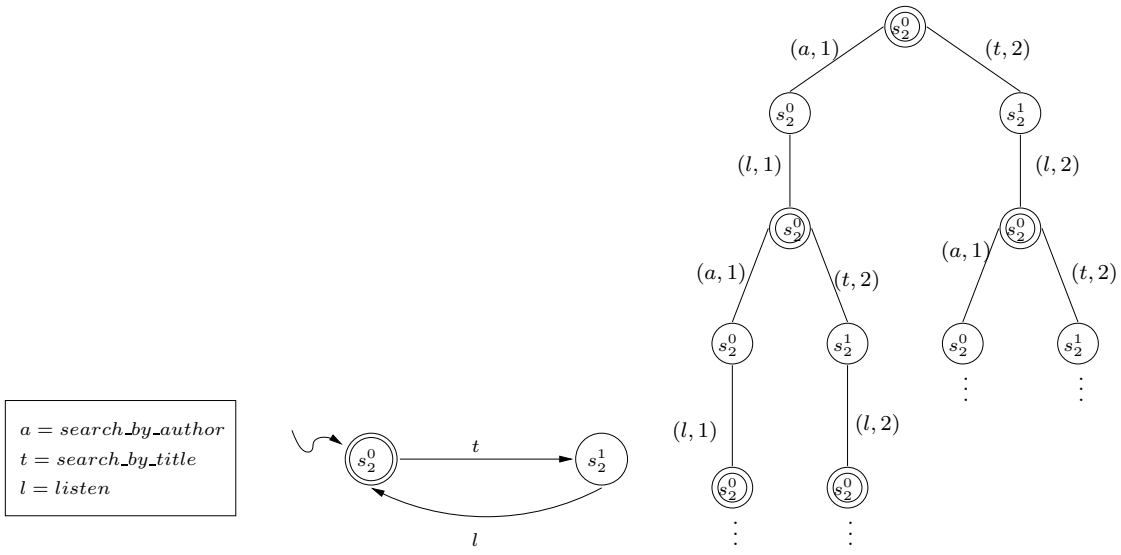
The mapping σ_2 from nodes of $T({E_0}^{int})$ to states of A_2 is defined as follows.

$\sigma_2(\varepsilon) = s_2^0$
$\sigma_2(a) = s_2^0$
$\sigma_2(t) = s_2^1$
$\sigma_2(a{\cdot}l) = \sigma_2(t{\cdot}l) = s_2^0$
$\sigma_2(a{\cdot}l{\cdot}a) = \sigma_2(t{\cdot}l{\cdot}a) = s_2^0$
$\sigma_2(a{\cdot}l{\cdot}t) = \sigma_2(t{\cdot}l{\cdot}t) = s_2^1$
$\sigma_2(a{\cdot}l{\cdot}a{\cdot}l) = \sigma_2(a{\cdot}l{\cdot}t{\cdot}l) = \sigma_2(t{\cdot}l{\cdot}a{\cdot}l) = \sigma_2(t{\cdot}l{\cdot}t{\cdot}l) = s_2^0$
...



(a) External schema A_1 of component e-Service E_1 (b) The mapping σ_1 on $T({E_0}^{int})$

Fig. 10. Composition of e-Services



(a) External schema A_2 of component e-Service E_2 (b) The mapping σ_2 on $T({E_0}^{int})$

Fig. 11. Composition of e-Services

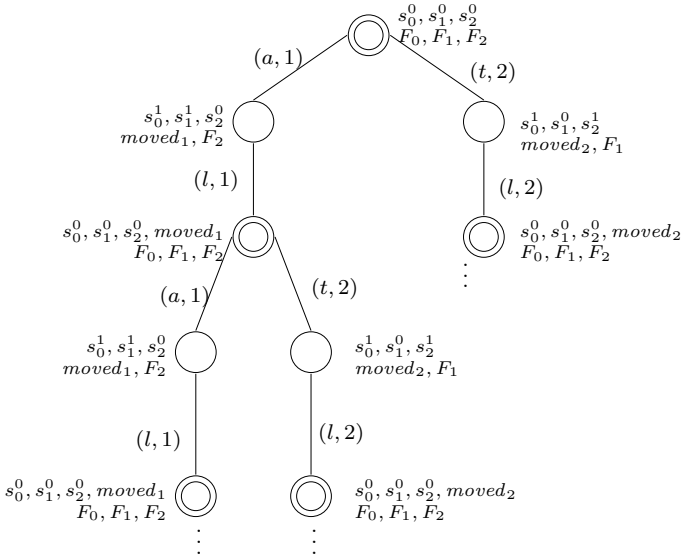


Fig. 12. Infinite model $\mathcal{I}$ for Φ.

Figure 11(b) represents the internal execution tree of E_0, where each node is labeled with the corresponding state of the automaton. σ_2 maps over s_2^1 the nodes of the tree that represent strings ending by t; it maps over s_2^0 the root of the tree and the nodes of the tree associated to strings ending by l or by a.

Given σ, σ_1 and σ_2, we define $\mathcal{I} = (\Delta^{\mathcal{I}}, \{a^{\mathcal{I}}\}_{a\in\Sigma}, \{P^{\mathcal{I}}\}_{P\in\mathcal{P}})$ of Φ as follows:

- $\Delta^{\mathcal{I}} = \{\varepsilon, a, t, a{\cdot}l, t{\cdot}l, a{\cdot}l{\cdot}a, a{\cdot}l{\cdot}t, t{\cdot}l{\cdot}a, t{\cdot}l{\cdot}t, a{\cdot}l{\cdot}a{\cdot}l, a{\cdot}l{\cdot}t{\cdot}l, t{\cdot}l{\cdot}a{\cdot}l, t{\cdot}l{\cdot}t{\cdot}l, \ldots\}$,
- $a^{\mathcal{I}} = \{(\varepsilon, a), (a{\cdot}l, a{\cdot}l{\cdot}a), (t{\cdot}l, t{\cdot}l{\cdot}a), \ldots\}$,
- $t^{\mathcal{I}} = \{(\varepsilon, t), (a{\cdot}l, a{\cdot}l{\cdot}t), (t{\cdot}l, t{\cdot}l{\cdot}t), \ldots\}$,
- $l^{\mathcal{I}} = \{(a, a{\cdot}l), (t, t{\cdot}l), (a{\cdot}l{\cdot}a, a{\cdot}l{\cdot}a{\cdot}l), (a{\cdot}l{\cdot}t, a{\cdot}l{\cdot}t{\cdot}l), (t{\cdot}l{\cdot}a, t{\cdot}l{\cdot}a{\cdot}l), (t{\cdot}l{\cdot}t, t{\cdot}l{\cdot}t{\cdot}l), \ldots\}$
- $(s_0^0)^{\mathcal{I}} = \{\varepsilon, a{\cdot}l, t{\cdot}l, a{\cdot}l{\cdot}a{\cdot}l, a{\cdot}l{\cdot}t{\cdot}l, t{\cdot}l{\cdot}a{\cdot}l, t{\cdot}l{\cdot}t{\cdot}l, \ldots\}$
- $(s_0^1)^{\mathcal{I}} = \{a, t, a{\cdot}l{\cdot}a, a{\cdot}l{\cdot}t, t{\cdot}l{\cdot}a, t{\cdot}l{\cdot}t, \ldots\}$
- $(s_1^0)^{\mathcal{I}} = \{\varepsilon, t, a{\cdot}l, t{\cdot}l, a{\cdot}l{\cdot}t, t{\cdot}l{\cdot}t, a{\cdot}l{\cdot}a{\cdot}l, a{\cdot}l{\cdot}t{\cdot}l, t{\cdot}l{\cdot}a{\cdot}l, t{\cdot}l{\cdot}t{\cdot}l, \ldots\}$
- $(s_1^1)^{\mathcal{I}} = \{a, a{\cdot}l{\cdot}a, t{\cdot}l{\cdot}a, \ldots\}$
- $(s_2^0)^{\mathcal{I}} = \{\varepsilon, a, a{\cdot}l, t{\cdot}l, a{\cdot}l{\cdot}a, t{\cdot}l{\cdot}a, a{\cdot}l{\cdot}a{\cdot}l, a{\cdot}l{\cdot}t{\cdot}l, t{\cdot}l{\cdot}a{\cdot}l, t{\cdot}l{\cdot}t{\cdot}l, \ldots\}$
- $(s_2^1)^{\mathcal{I}} = \{t, a{\cdot}l{\cdot}t, t{\cdot}l{\cdot}t, \ldots\}$
- $moved_1^{\mathcal{I}} = \{a, a{\cdot}l, a{\cdot}l{\cdot}a, t{\cdot}l{\cdot}a, a{\cdot}l{\cdot}a{\cdot}l, t{\cdot}l{\cdot}a{\cdot}l, \ldots\}$
- $moved_2^{\mathcal{I}} = \{t, t{\cdot}l, a{\cdot}l{\cdot}t, t{\cdot}l{\cdot}t, a{\cdot}l{\cdot}t{\cdot}l, t{\cdot}l{\cdot}t{\cdot}l, \ldots\}$
- $F_0^{\mathcal{I}} = \{\varepsilon, a{\cdot}l, t{\cdot}l, a{\cdot}l{\cdot}a{\cdot}l, a{\cdot}l{\cdot}t{\cdot}l, t{\cdot}l{\cdot}a{\cdot}l, t{\cdot}l{\cdot}t{\cdot}l, \ldots\}$
- $F_1^{\mathcal{I}} = \{\varepsilon, t, a{\cdot}l, t{\cdot}l, a{\cdot}l{\cdot}t, t{\cdot}l{\cdot}t, a{\cdot}l{\cdot}a{\cdot}l, a{\cdot}l{\cdot}t{\cdot}l, t{\cdot}l{\cdot}t{\cdot}l, t{\cdot}l{\cdot}t{\cdot}l, \ldots\}$
- $F_2^{\mathcal{I}} = \{\varepsilon, a, a{\cdot}l, t{\cdot}l, a{\cdot}l{\cdot}a, t{\cdot}l{\cdot}a, a{\cdot}l{\cdot}a{\cdot}l, a{\cdot}l{\cdot}t{\cdot}l, t{\cdot}l{\cdot}a{\cdot}l, t{\cdot}l{\cdot}t{\cdot}l, \ldots\}$

Figure 12 shows that $\mathcal{I}$ is a model for the formula Φ[9]. Each node of the tree is associated with the propositions in $\mathcal{P}$ that hold in that node, according to $\mathcal{I}$. For example, consider the root: $\mathcal{I}$ imposes that $s_0^0 \wedge s_1^0 \wedge s_2^0 \wedge F_0 \wedge F_1 \wedge F_2$ holds in ε. Note that for sake of readability, in the figure we have associated to each node 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 root: we have arbitrarily chosen their value to be `false`. Finally, note that $\mathcal{I}$ is not finite (the figure shows only a portion of the tree).

Because of the small model property, Φ admits a finite model $\mathcal{I}_f$, shown in Figure 13 as a FSM.

The finite model $\mathcal{I}_f$ induces mappings σ^f, σ_1^f and σ_2^f from its states to states of the automata representing the external schema of the target e-Service and of the component ones.

[9] The action labeling on edges, of course, is not part of the model: we report it for readability.

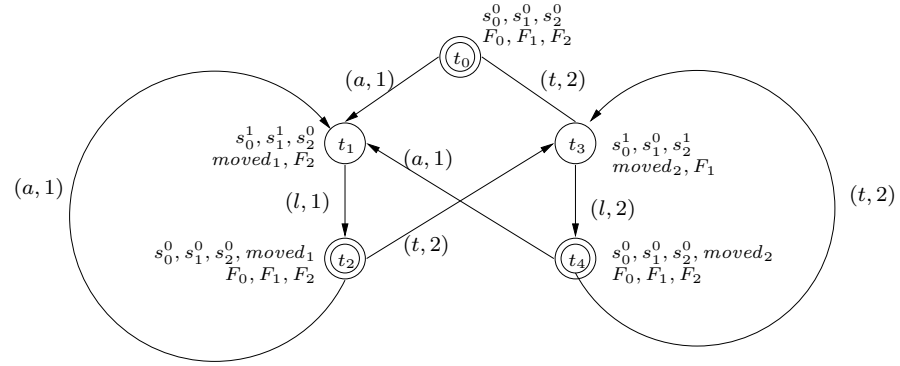


Fig. 13. Finite model $\mathcal{I}_f$ for Φ.

$\sigma^f(t_0) = \sigma^f(t_2) = \sigma^f(t_4) = s_0^0$
$\sigma^f(t_1) = \sigma^f(t_3) = s_0^1$
$\sigma_1^f(t_0) = \sigma_1^f(t_2) = \sigma_1^f(t_3) = \sigma_1^f(t_4) = s_1^0$
$\sigma_1^f(t_1) = s_1^1$
$\sigma_2^f(t_0) = \sigma_2^f(t_1) = \sigma_2^f(t_2) = \sigma_2^f(t_4) = s_2^0$
$\sigma_2^f(t_3) = s_2^1$

Given σ^f, σ_1^f and σ_2^f, we can define $\mathcal{I}_f = (\Delta_f^{\mathcal{I}}, \{a^{\mathcal{I}_f}\}_{a\in\Sigma}, \{P^{\mathcal{I}_f}\}_{P\in\mathcal{P}})$ of Φ as follows:

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

Given the finite model $\mathcal{I}_f = (\Delta_f^{\mathcal{I}}, \{a^{\mathcal{I}_f}\}_{a\in\Sigma}, \{P^{\mathcal{I}_f}\}_{P\in\mathcal{P}})$ of Φ, we define the 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 e-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:

$\delta_c(t_0, a) = t_1$	$\delta_c(t_2, a) = t_1$
$\delta_c(t_0, t) = t_3$	$\delta_c(t_2, t) = t_3$
$\delta_c(t_1, l) = t_2$	$\delta_c(t_4, a) = t_1$
$\delta_c(t_3, l) = t_4$	$\delta_c(t_4, t) = t_3$

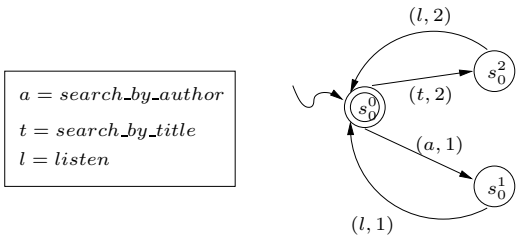


Fig. 14. Minimal FSM associated to $T({E_0}^{int})$.

– 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 FSM associated to the tree representing the infinite model is shown in Figure 14. It is easy to see that it does not represent a model for Φ since, for instance, state t_0 is associated to both $moved_1$ and $\neg moved_1$.

7 Related Work

Up to now, research on *e*-Services has mainly concentrated on three issues, namely *(i)* service description and modeling, *(ii)* service discovery (e.g., [24]) and *(iii)* service composition, including synthesis and orchestration.

Current research in description and modeling of *e*-Services is mainly founded on the work on workflows, which model business processes as sequences of (possibly partially) automated activities, in terms of data and control flow among them (e.g., [22, 14]). In [18] *e*-Services are represented as statecharts, and in [7], an *e*-Service is modeled as a Mealy machine, with input and output messages, and a queue is used to buffer messages that were received but not yet processed.

In our paper, we model *e*-Services as finite state machines, even if we do not consider communication delays and therefore any concept of message queuing is not taken into account. Indeed, from the survey of [13], it stems that the most practical approaches for modeling and describing *e*-Services are the ones based on specific forms of state machines. Additionally, our model of *e*-Service is oriented towards representing the interactions between a client and an *e*-Service. Therefore, our focus is on action sequences, rather than on message sequences as in [7], or on actions with input/output parameters as in [16].

As far as orchestration, it requires that the composite *e*-Service is specified in a precise way, considering both the specification of how various component *e*-Services are linked and the internal process flow of the component one. In [13], different technologies, standards and approaches for specification of composite *e*-Services are considered, including BPEL4WS, BPML, AZTEC, etc. Reference [13] identifies three different kinds of composition: *(i)* peer-to-peer, in which the individual *e*-Services are equals, *(ii)* the mediated approach, based on a hub-and-spoke topology, in which one service is given the role of process mediator, and *(iii)* the brokered approach, where process control is centralized but data can pass between component *e*-Services. With respect to such a classification, the approach proposed in this paper belongs to the mediated one.

Also most of other research works [9, 23, 17] can be classified into the mediated approach to composition. Conversely in [10] the enactment of a composite *e*-Service is carried out in a decentralized way, through peer-to-peer interactions.

The *DAML-S Coalition* [2] is defining a specific ontology and a related language for *e*-Services, with the aim of composing them in automatic way. In [25] the issue of service composition is addressed, in order to create composite services by re-using, specializing and extending existing ones; in [16, 19] composition of *e*-Services is addressed by using GOLOG and providing a semantics of the composition based on Petri Nets. In [1] a way of composing *e*-Services is presented, based on planning under uncertainty and constraint satisfaction techniques, and a request language, to be used for specifying client goals, is proposed. *e*-Service composition is indeed a form of program synthesis as is planning. The main conceptual difference is that, while in planning we typically are interested in synthesizing a *new* sequences of actions (or more generally a program, i.e., an execution tree) that achieves the client goal, in *e*-Service composition, we try to obtain (the execution tree of) the target *e*-Service by *reusing* in a suitable way fragments of the executions of the component *e*-Services.

In [7], the interplay between a composite *e*-Service (global) and component ones (local) is considered. The authors represent *e*-Services as FSMs and show that composite *e*-Services may no longer be a FSM in presence of unexpected behavior.

8 Conclusions

The main contribution of this paper wrt research on service oriented computing is in tackling *simultaneously* the following issues: *(i)* presenting a formal model where the problem of *e*-Service composition is precisely characterized, *(ii)* providing techniques for computing *e*-Service composition in the case of *e*-Services represented by finite state machines, and *(iii)* providing a computational complexity characterization of the algorithm for automatic composition.

In the future we plan to extend our work both in practical and theoretical directions. On one side, we are developing a Description Logic based prototype system that implements the composition technique presented in the paper. Such system will enable us to test how the complexity of composition in our framework impacts real world applications. On the theoretical side, we will address open issues such as the characterization of a lower bound for the complexity of the composition problem. Additionally, in the proposed framework, we have made the fundamental assumption that one has complete knowledge on the *e*-Services belonging to a community, in the form of their external and internal schema. We also assumed that a client gives a very precise specification (i.e., the external schema) of an *e*-Service he wants to have realized by a community. In particular, such a specification does not contain forms of "don't care" nondeterminism. Both such assumptions can be relaxed, and this leads to a development of the proposed framework that is left for further research. Finally, we plan to extend our setting, by also considering the presence of communication delays and of an unbounded number of active instances.

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 *Proceedings of the 3rd VLDB International Workshop on Technologies for* e-*Services (VLDB-TES 2002)*, Hong Kong, China, 2002.
2. 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 *Proceedings of the 1st International Semantic Web Conference (ISWC 2002)*, Chia, Sardegna, Italy, 2002.
3. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.
4. Mordechai Ben-Ari, Joseph Y. Halpern, and Amir Pnueli. Deterministic propositional dynamic logic: Finite models, complexity, and completeness. *Journal of Computer and System Sciences*, 25:402–417, 1982.
5. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. A foundational vision of *e*-Services. In *Proceedings of the CAiSE 2003 Workshop on Web Services, e-Business, and the Semantic Web (WES 2003)*, Velden, Austria, 2003.
6. D. Berardi, F. De Rosa, L. De Santis, and M. Mecella. Finite state automata as conceptual model for e-services. In *Proc. of the IDPT 2003 Conference*, 2003. To appear.
7. T. Bultan, X. Fu, R. Hull, and J. Su. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. In *Proceedings of the WWW 2003 Conference*, Budapest, Hungary, 2003.

8. Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Daniele Nardi. Reasoning in expressive description logics. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 23, pages 1581–1634. Elsevier Science Publishers (North-Holland), Amsterdam, 2001.
9. F. Casati and M.C. Shan. Dynamic and adaptive composition of *e*-Services. *Information Systems*, 6(3), 2001.
10. M.C. Fauvet, M. Dumas, B. Benatallah, and H.Y. Paik. Peer-to-Peer Traced Execution of Composite Services. In *Proceedings of the 2nd VLDB International Workshop on Technologies for* e-*Services (VLDB-TES 2001)*, Rome, Italy, 2001.
11. Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18:194–211, 1979.
12. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. The MIT Press, 2000.
13. R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: A Look Behind the Curtain. In *Proceedings of the PODS 2003 Conference*, San Diego, CA, USA, 2003.
14. E. Kafeza, D.K.W. Chiu, and I. Kafeza. View-based Contracts in an *e*-Service Cross-Organizational Workflow Environment. In *Proceedings of the 2nd VLDB International Workshop on Technologies for* e-*Services (VLDB-TES 2001)*, Rome, Italy, 2001.
15. Dexter Kozen and Jerzy Tiuryn. Logics of programs. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science — Formal Models and Semantics*, pages 789–840. Elsevier Science Publishers (North-Holland), Amsterdam, 1990.
16. S. McIlraith and T. Son. Adapting Golog for Composition of Semantic Web Services. In *Proceedings of the 8th International Conference on Knowledge Representation and Reasoning (KR 2002)*, Toulouse, France, 2002.
17. M. Mecella and B. Pernici. Building Flexible and Cooperative Applications Based on *e*-Services. Technical Report 21-2002, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Roma, Italy, 2002. (available on line at: `http://www.dis.uniroma1.it/~mecella/publications/mp_techreport_212002.pdf`).
18. M. Mecella, B. Pernici, and P. Craca. Compatibility of *e*-Services in a Cooperative Multi-Platform Environment. In *Proceedings of the 2nd VLDB International Workshop on Technologies for* e-*Services (VLDB-TES 2001)*, Rome, Italy, 2001.
19. S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proceedings of the 11th International Conference on World Wide Web*, Hawaii, USA, 2002.
20. Mike P. Papazoglou. Agent-Oriented Technology in Support of e-Business. *Communications of the ACM*, October 2003. To appear.
21. T. Pilioura and A. Tsalgatidou. *e*-Services: Current Technologies and Open Issues. In *Proceedings of the 2nd VLDB International Workshop on Technologies for* e-*Services (VLDB-TES 2001)*, Rome, Italy, 2001.
22. H. Schuster, D. Georgakopoulos, A. Cichocki, and D. Baker. Modeling and Composing Service-based and Reference Process-based Multi-enterprise Processes. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE 2000)*, Stockholm, Sweden, 2000.
23. G. Shegalov, M. Gillmann, and G. Weikum. XML-enabled workflow management for *e*-Services across heterogeneous platforms. *Very Large Database J.*, 10(1), 2001.
24. W.J. van den Heuvel, J. Yang, and M.P. Papazoglou. Service Representation, Discovery and Composition for *e*-Marketplaces. In *Proceedings of the 9th International Conference on Cooperative Information Systems (CoopIS 2001)*, Trento, Italy, 2001.
25. J. Yang and M.P. Papazoglou. Web Components: A Substrate for Web Service Reuse and Composition. In *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE'02)*, Toronto, Canada, 2002.
26. J. Yang, W.J. van den Heuvel, and M.P. Papazoglou. Tackling the Challenges of Service Composition in *e*-Marketplaces. In *Proceedings of the 12th International Workshop on Research Issues on Data Engineering: Engineering E-Commerce/E-Business Systems (RIDE-2EC 2002)*, San Jose, CA, USA, 2002.

A Deterministic Propositional Dynamic Logic

Propositional Dynamic Logics (PDLs) are a family of modal logics specifically developed for reasoning about computer programs [15]. They capture the properties of the interaction between programs and propositions that are independent of the domain of computation. In this appendix, we provide a brief overview of a logic of this family, namely Deterministic Propositional Dynamic Logic (DPDL). More details can be found in [12].

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:

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

where P is an atomic proposition in $\mathcal{P}$, r is a regular expression over the set of actions in $\mathcal{A}$, 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 exeute 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 PDLs (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 PDL formulas (and modal logics) is defined over a structure, namely a Kripke structure.

The semantics of a DPDL formula is based on a 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* $a^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ from elements of $\Delta^{\mathcal{I}}$ to elements of $\Delta^{\mathcal{I}}$, each of which denotes the state transitions caused by the atomic program a[10]; $P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ 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:

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

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:

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

[10] Note that the determinism of the Kripke structure derives from the fact that $a^{\mathcal{I}}$ assigns to each state in $\Delta^{\mathcal{I}}$ a unique successor state.

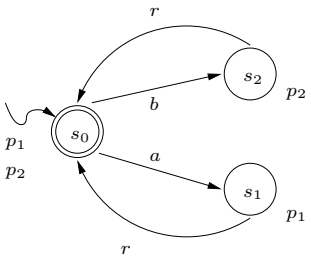


Fig. 15. Kripke structure for Example 11.

Example 11. Let $\mathcal{P} = \{p_1, p_2\}$ be the set of atomic propositions, let $\mathcal{A} = \{a, b, r\}$ be the set of atomic actions and let $\mathcal{I} = (\Delta^{\mathcal{I}}, \{a^{\mathcal{I}}\}_{a\in\mathcal{A}}, \{P^{\mathcal{I}}\}_{P\in\mathcal{P}})$ be the Kripke structure shown in Figure A with:

$$\begin{aligned}\Delta^{\mathcal{I}} &= \{s_0, s_1, s_2\}\\ \{a^{\mathcal{I}}\} &= \{(s_0, s_1)\}\\ \{b^{\mathcal{I}}\} &= \{(s_0, s_2)\}\\ \{r^{\mathcal{I}}\} &= \{(s_1, s_0), (s_2, s_0)\}\\ \{p_1^{\mathcal{I}}\} &= \{s_0, s_1\}\\ \{p_2^{\mathcal{I}}\} &= \{s_0, s_2\}\end{aligned}$$

It is easy to see that in this structure, $s_0 \models [a]p_1 \wedge [b]p_2 \wedge [r]\texttt{false}$, $s_1 \models [r](p_1 \wedge p_2)$, and $s_2 \models [r](p_1 \wedge p_2)$.

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 PDLs. Such a set of formulas is given by the Fischer-Ladner closure [11].

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. 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 [4]:

Theorem 4. *Satisfiability in DPDL is EXPTIME-complete.*