Automatic Composition of *e*-Services

Daniela Berardi

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

Supervisor: Maurizio Lenzerini
Thesis Committee: Maurizio Lenzerini, Giuseppe De Giacomo, and Massimo Mecella

1 Introduction

Service Oriented Computing (SOC [35]) 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 [36]), 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 [42]. In this Ph.D. thesis, 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 issue, 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 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 Ph.D. thesis is on automatic composition synthesis. In order to address this issue in an effective and well-founded way, as a first contribution, in [8, 9] we propose a general

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

formal framework for representing *e*-Services. As a second contribution, in [8] we present 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. In [10, 7] we propose an alternative (though deeply related) approach, that is based on characterizing *e*-Services and *e*-Service composition in terms of Reasoning about Actions.

The rest of this research report is organized as follows. In Section 2 we consider the state-of-the-art of the research on *e*-Services and in Section 3 we highlight open issues and contextualize our contribution by presenting how our research overcome such problems. In Sections 4, 5 and 6 we define our general formal framework, and in Section 7 we define the problem of composition synthesis in such a framework. In Section 8 we specialize the general framework to the case where *e*-Services are specified by means of finite state machines, and in Section 9 we present an EXPTIME algorithm for automatic *e*-Service composition in the specialized framework. In Section 10 we instantiate our general framework to a specific formalism for Reasoning about Actions, namely Situation Calculus, and show equivalent algorithms to compute *e*-Service composition. Finally, in Section 11 we discuss future work.

2 State-of-the-art

Up to now, research on *e*-Services has mainly concentrated on the issues of *(i)* service description and modeling and of *(ii)* 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., [38, 27]). In [33] *e*-Services are represented as statecharts, and in [12], 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.

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 [26], different technologies, standards and approaches for specification of composite *e*-Services are considered, including BPEL4WS, BPML, AZTEC, etc. Reference [26] 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. Most of research works [14, 39, 32] can be classified into the mediated approach to composition. Conversely in [21] 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 [41] the issue of service composition is addressed, in order to create composite services by re-using, specializing and extending existing ones; in [31, 34] 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 [12], 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.

Finally, it is worth noting that theoretical investigations on *e*-Service composition have been explicitly or implicitly addressed in various forms by several research areas, including the following ones: *(i)* Workflows [14, 27, 39], from which *e*-Service composition derives most of its roots; *(ii)* Databases (see, e.g., [40]), where query rewriting techniques for Data Integration may be seen as simple forms of composition of simple "data-access" *e*-Services, which can described by an atomic action with query/results as input/output parameters; *(iii)* Software Engineering [43, 23], since several works on architecture design based on components and objects have addressed theoretical issues related to software composition, as well as those related to program synthesis and verification; *(iv)* Artificial Intelligence [31, 34, 1], where *e*-Services are considered as atomic actions with input/output parameters, and agent-based technologies and planning techniques, supported by domain ontologies, have been advocated as basic tools for action and process composition: *(v)* Theoretical Computer Science, and in particular, Language and Automata Theory. Moreover, many standards (e.g., WSDL [3], BPEL4WS [15], ebXML [19, 20, 18], WSCI [5], just to refer to the most widespread) are providing an effective infrastructure for *e*-Service composition, but they lack clear semantics, formal analysis of their properties, guidelines and methodologies on how to apply them and effectively use them to deliver composite *e*-Service. However, a global, coherent, flexible and comprehensive vision, allowing for comparing such different approaches, is still lacking. Indeed, each area focuses on a particular feature of *e*-Services and therefore, *e*-Service composition is tackled in heterogeneous ways: in some context composition is automatic, in other manual, in others it is restricted to the enactment of the composite *e*-Service without considering the synthesis of the composite service, etc. Of course, such a plethora of approaches highly depends on the expressiveness and effectiveness of the various languages used to describe services.

3 Motivations

Although an enormous interest is moving around *e*-Services, several aspects related to *e*-Service composition, and as an aside, to *e*-Service description, including foundational ones, still remain to be clarified (see [26] for a survey on different approaches to service oriented computing).

- An agreed comprehension of what an *e*-Service is, in an abstract and general fashion, is still lacking. An *e*-Service is mostly considered simply as a set of operations with input and output parameters, with no constraints on their invocation order. Only recently, few proposals are advocating that an *e*-Service should also be characterized by its behavior, i.e., intuitively, by the set of admissible invocation sequences of operations. Therefore, no general and common framework exists that contextualizes *e*-Services and *e*-Service composition.
- There does not exist a consolidated formal definition of *e*-Service composition. Additionally, to the best of our knowledge, no approach to *e*-Service composition exists that explicitly takes the client's need into account.
- Due to the absence of a common vision, it is extremely difficult to compare the various approaches to composition. As a notable example, results on computational complexity of both

the problem of *e*-Service composition, and the algorithms for composition synthesis are still lacking, and this inhibits practical and commercial developments of tools for composition.
- A clear and consolidated awareness of the relations between languages and tools for describing *e*-Services and composition techniques is not present.
- A consolidated characterization of an adequate set of operators for *e*-Service composition is lacking, as well as, a definition and classification of possible languages for composition.
- There does not exist a deep analysis of the possible types of composition, and their properties.

The aim of this Ph.D. thesis is to define a formal and comprehensive framework for the characterization and the theoretical investigation of the problem of *e*-Service composition.

Although several papers have been already published that discuss either a formal model of *e*-Services (even more expressive than ours, see e.g., [12]), or propose algorithms for computing composition (e.g., [34]), to the best of our knowledge, the research done till now and reported here is the first one tackling simultaneously the following issues:

- Presenting a formal framework where *e*-Services are clearly defined and the problem of automatic *e*-Service composition is precisely characterized. Although simplified in several aspects, our framework is general, comprehensive and coherent enough to accommodate various visions on *e*-Services and *e*-Service composition. Additionally, it is flexible and robust, so that changes in the vision on *e*-Service composition can be reflected on it with few adjustments.
- Providing techniques for computing *e*-Service composition in special but quite significant cases (finite state *e*-Services). In particular, this is the first algorithm for automatic *e*-Service composition that also explicitly takes into account the client's needs.
- Providing a computational complexity characterization of the algorithm for automatic composition.

However, several open issues remain to be solved and many possible extensions to our framework may be taken into account. Additionally, not all aspects highlighted above have been tackled yet. Doubtless, the research done this year has constituted a first step towards the definition of a theoretical framework for *e*-Services and *e*-Service composition.

4 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

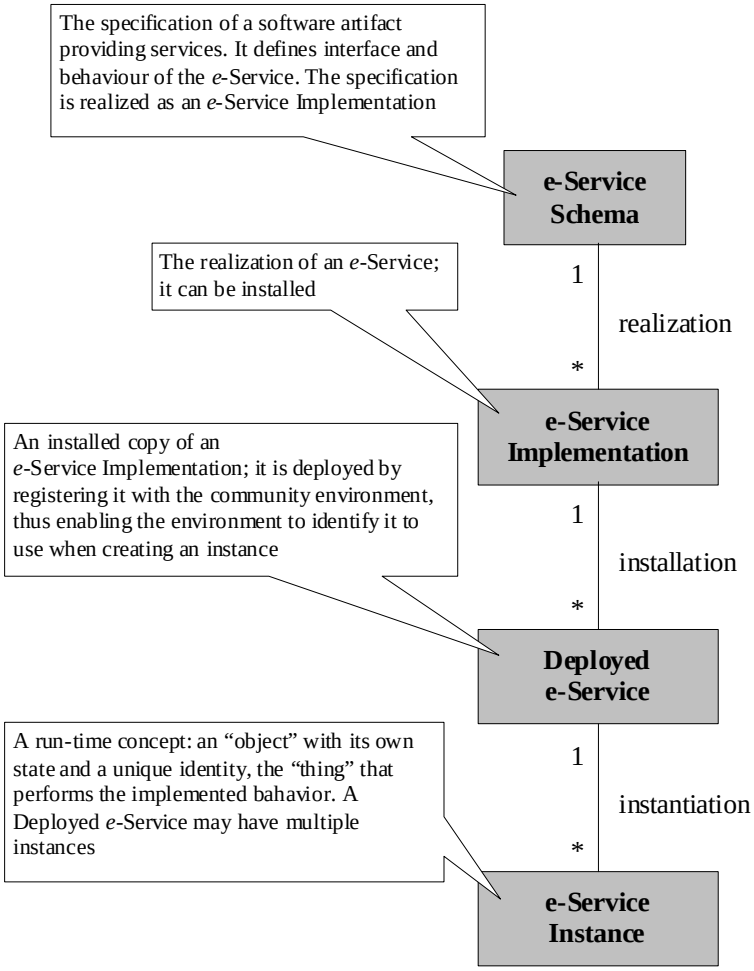


Fig. 1. *e*-Service facets

the scope of our research. 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.

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 our research.

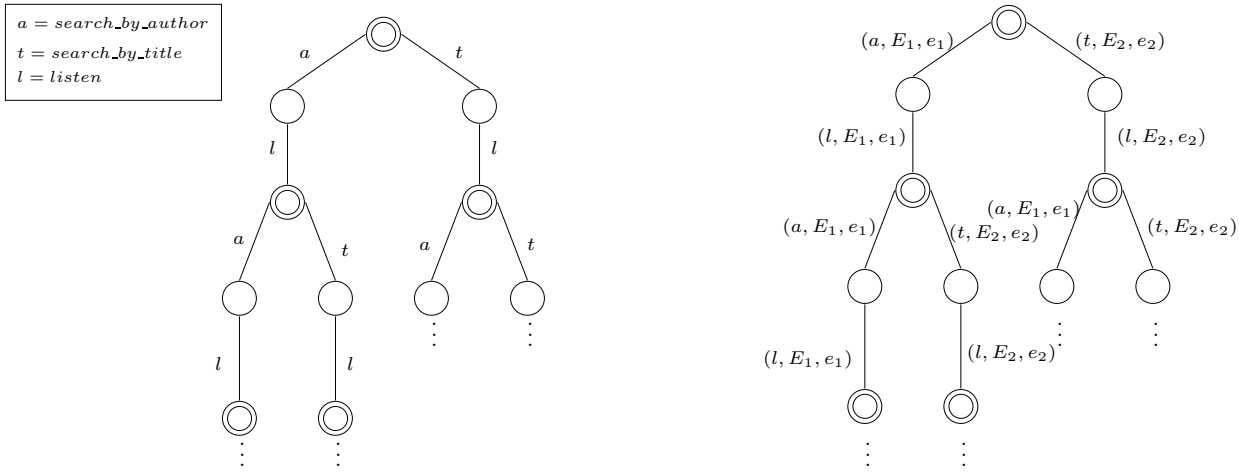
In the following sections we formally describe how the e-Services of a community are specified, through the notion of e-Service schema, and how they are executed, through the notion of e-Service instance.

5 *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.

5.1 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



(a) External tree

(b) Internal tree

Fig. 2. Execution trees of e-Service E_0

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 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]. Observe that non final states are common in interactive e-Services (for humans) over the web. There, however, it is always possible to abort the entire transaction. Here, we consider the abortion mechanism as orthogonal to the e-Service specification.

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. 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 actions with input/output parameters.

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(a) 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

[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.

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. □

5.2 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[5]. 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 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.

[5] Note that, in general, an action can be executed by one or more *e*-Service instances. The opportunity of allowing more than one component *e*-Service to execute the same action is important in specific situations, as the one reported in [10].

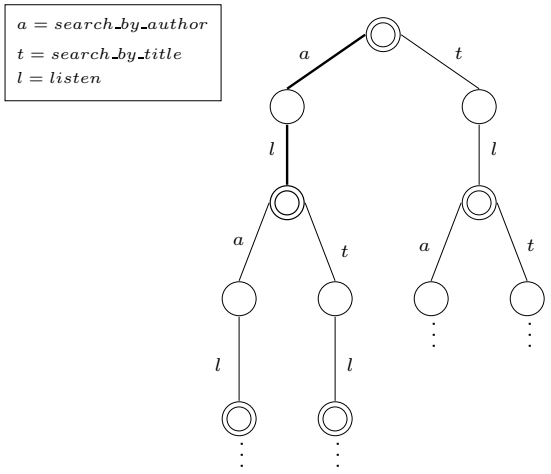


Fig. 3. External view of an *e*-Service instance

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 2(b)[6] shows (a portion of) an (infinite) internal execution tree, conforming to the external execution tree of *e*-Service E_0 shown in Figure 2(a), where all the actions are delegated to *e*-Services of the community. In particular, the execution of `search_by_author` action and its subsequent `listen` action are delegated to instance e_1 of *e*-Service E_1, and the execution of `search_by_title` action and its subsequent `listen` action to instance e_2 of *e*-Service E_2. □

6 *e*-Service Instances

In order to be executed, a deployed *e*-Service has to be activated, i.e., necessary resources need to be allocated. An *e*-Service instance represents such an *e*-Service running and interacting with its client.

From an abstract point of view, a running instance corresponds to an execution tree with a highlighted node, representing the "current position", i.e., the point reached by the execution. The path from the root of the tree to the current position is the run of the *e*-Service at a certain point, while the execution (sub-)tree having as root the current position describes the behavior of what remains of the *e*-Service once the current position is reached.

Formally, an *e*-Service instance is characterized by:

- an *instance identifier*,
- an *external view* of the instance, which is an external execution tree with a current position,
- an *internal view* of the instance, which is an internal execution tree with a current position.

Example 3. Figure 3 shows the external view of an instance of the *e*-Service E_0 of Figure 2 (a). The sequence of actions executed so far and the current position on the execution tree are shown in thick lines. It represents a snapshot of an execution in which a client searched for an mp3 by specifying its author(s), listened to it, and reached a point where he has to choose whether *(i)* performing

[6] 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 $i = 1, 2$.

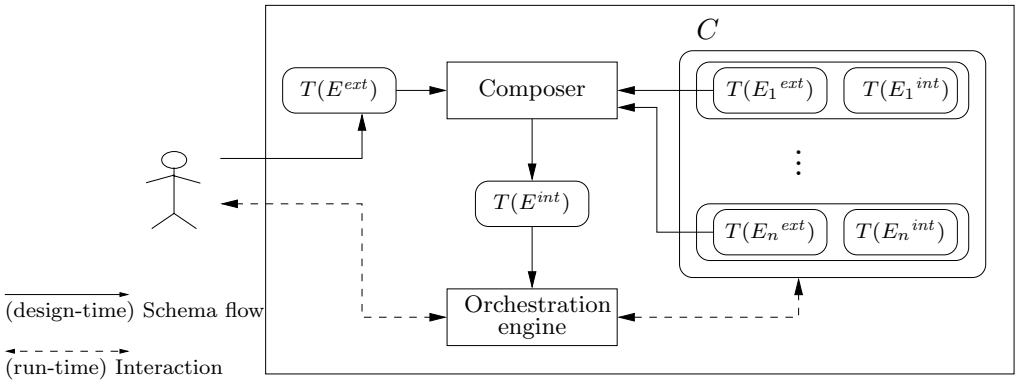


Fig. 4. *e*-Service Integration System

another `search_by_author` action, *(ii)* performing a `search_by_title` action, or *(iii)* terminating the *e*-Service (since the current position corresponds to a final node). □

The internal view of an *e*-Service instance additionally maintains information on which *e*-Service instances execute which actions. At each point of the execution there may be several other active instances of *e*-Services that cooperate with the current one, each identified by its instance identifier.

7 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, specified as an external execution tree $T(E^{ext})$, 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}[7] of E, specified as an internal execution tree $T(E^{int})$, and that is a composition of E wrt the community C. Then, following $T(E^{int})$, an *orchestration engine* activates an (internal) instance of E, and

[7] If at least one exists.

orchestrates the different available e-Services, by activating and interacting with their external views, 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.

8 *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).

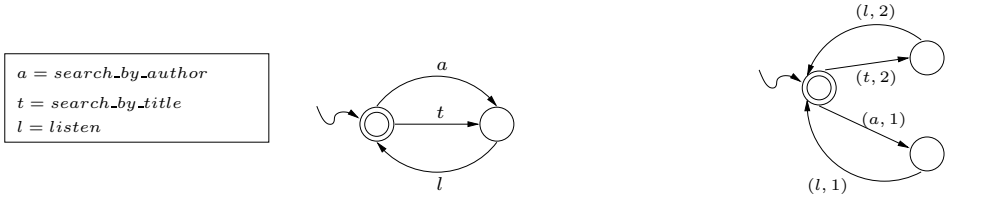
The class of e-Services that can be captured by FSMs are of particular interest. This class allows us to address an interesting set of e-Services, that are able to carry on rather complex interactions with their clients, performing useful tasks. Indeed, several papers in the e-Service literature adopt FSMs as the basic model of exported behavior of e-Services [26, 12, 11]. 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. Since the number of e-Services in a community is finite, 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.

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:



(a) External FSM (b) Internal MFSM

Fig. 5. *e*-Service specification as FSM

- 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 $\sigma(x) \in F_E$.

Figure 5(a) shows a FSM that is a specification for the external execution tree of Figure 2(a). Note that in general there may be several FSMs that may serve as such a specification.

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*-Services 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`.

As far as internal schemas, for an *e*-Service E, we are interested in those having 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 the *e*-Service E is in the 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.

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:

- 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 defined, $x \cdot a$ 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 $\sigma^{int}(x) \in F_E^{int}$.

As an example, Figure 5(b) shows a MFSM that is a specification for an internal execution tree that conforms to the external execution tree specified by the FSM of Figure 5(a). Indeed the MFSM in the figure compactly represents the e-Service whose internal execution tree is shown in Figure 2(b). In general, an external schema specified as FSM and its corresponding internal schema specified as MFSM may have different structures, as the example shows.

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 report, we do not go into the details of these problems, and instead we concentrate on composition.

9 Automatic *e*-Service Composition

In this section, 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 [28]. We refer to Appendix A for a brief tutorial on DPDL.

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 FSMs $A_1, \ldots, A_n$ 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, $j = 0, \ldots, n$, that is true if 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, $\mathcal{A} = \Sigma$). 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.
– 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, conditioned to the fact that the component A_i is actually required to make a transition a in the composition.
  - $[u](s \rightarrow [a]\neg moved_i)$ 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.

– 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 $[u]\cdot$.
  - $[u](\langle a\rangle \texttt{true} \rightarrow [a] \bigvee_{i=1,\ldots,n} moved_i)$, 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.

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 the size of A_0, $A_1, \ldots, A_n$. 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 $E_1, \ldots, E_n$, and 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 result.

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 $E_1, \ldots, E_n$. □

In Appendix B a detailed example is provided, that explains the composition synthesis algorithm step by step.

From a practical point of view, because of the correspondence between Propositional Dynamic Logics (which DPDL belongs to) and Description Logics (DLs [13]), one can use current highly optimized DL-based systems [4, 24][8] 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.

[8] In fact, current DL-based 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.

10 *e*-Service Composition in Situation Calculus

We address the problem of computing *e*-Service composition in our setting by following another different but equivalent approach. Specifically, we use formalisms developed for Reasoning about Actions to represent *e*-Services, and show that again we can use logical reasoning, and in particular, satisfiability, to characterize the problem of *e*-Service composition. As before, we focus on *e*-Services whose execution trees have a finite representation and assume that at most one instance of an *e*-Service in the community can be involved in the internal execution tree of a composite *e*-Service. Instead, again, how to deal with an unbounded number of instances remains open for future work.

There are many possible action languages that can be used for representing *e*-Services (including some tightly related to DL [17, 29]). Here we focus on Reiter's Situation Calculus Basic Action Theories [37], which are widely known and allow us to concentrate on the aspects specific to our problem. Since we aim at actually computing the compositions we will deal with the propositional variant of the Situation Calculus (in which fluents are propositions).

We will not go over the Situation Calculus [30] here, except to note the following components: there is a special constant S_0 used to denote the *initial situation*, namely that situation in which no actions have yet occurred; there is a distinguished binary function symbol do, where $do(a, s)$ denotes the successor situation to s resulting from performing the action a; propositions whose truth values vary from situation to situation are called (propositional) *fluents*, and are denoted by predicate symbols taking a situation term as their last argument; and there is a special predicate $Poss(a, s)$ used to state that action a is executable in situation s. Within this language, we can formulate domain theories that describe how the world changes as the result of the available actions. One possibility are Reiter's Basic Action Theories, which have the following form [37]:

- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action a, of the form $\forall s.Poss(a, s) \equiv \Psi_a(s)$, where $\Psi_a(s)$ is a Situation Calculus formula (uniform in s) with s as the only free variable and in which $Poss$ does not appear.
- Successor-state axioms, one for each fluent F, of the form $\forall a, s.F(do(a, s)) \equiv \Phi_F(a, s)$, where $\Phi_F(a, s)$ is a Situation Calculus formula (uniform in s) with a and s as the only free variables. These axioms take the place of effect axioms, but also provide a solution to the frame problem.
- Unique names axioms for the primitive actions plus some foundational, domain independent axioms.

In order to characterize composition in this setting, we first show how a Basic Action Theory can represent the external execution tree of an *e*-Service. We represent the external schema E^{ext} of an *e*-Service E as a Basic Action Theory Γ, where each action is represented by a Situation Calculus action. Γ includes among its fluents a special fluent *Final*, denoting that the *e*-Service execution can stop in that situation. Also, Γ fully specifies the value of each fluent in the initial situation S_0. Technically, this means that we have complete information on the initial situation, and, because of the action precondition and successor-state axioms, we have complete information in every situation.

Observe that the fluents used in Γ have a meaning only wrt to the *e*-Service community, since they are not attached in any way to the actual *e*-Service instance the client interacts with. In contrast, actions represent interactions meaningful both to the client and the *e*-Service instance.

Intuitively, the part of the situation tree [37] formed only by the actions that are possible (as specified by $Poss$) directly corresponds to the external execution tree $T(E^{ext})$ of the *e*-Service, where the final nodes are the situations in which *Final* is true. To formally define such an execution

tree, we first inductively define a function $n(\cdot)$ from situations to sequences of actions union a special value *undef*:

- $n(S_0) = \varepsilon$;
- $n(do(a, s)) = n(s) \cdot a$ if $n(s) \neq undef$ and $Poss(a, s)$ holds;
- $n(do(a, s)) = undef$ otherwise.

The execution tree $T(E^{ext})$ generated by Γ is defined over the set of nodes $\{n(s) \mid n(s) \neq undef\}$, such that a node $n(s)$ is final if and only if $Final(s)$ holds. It is easy to check that $T(E^{ext})$ is indeed an execution tree.

Next, we turn to the problem of characterizing *e*-Service composition. Let $\Gamma_1, \ldots, \Gamma_n$, be the theories for the component *e*-Services, and let Γ_0 be the theory for the target *e*-Service E_0. The basic idea is to represent which *e*-Services are executed when an action of the target *e*-Service is performed. We do this by means of special predicates $Step_i(a, s)$, denoting that the active instance of *e*-Service E_i executes action a in situation s. Formally, we construct a Situation Calculus theory Γ_C formed by the union of the axioms below:

- Γ_0;
- Γ_i', for $i = 1, \ldots, n$, where Γ_i' is obtained from Γ_i:
  1. by renaming each fluent F, including *Final*, to F_i;
  2. by renaming *Poss* to $Poss_i$;
  3. by modifying the successor-state axioms as follows:
     $\forall a, s. F_i(do(a, s)) \equiv (Step_i(a, s) \wedge \Phi_{F_i}(a, s)) \vee (\neg Step_i(a, s) \wedge F_i(s))$;
- $\forall a, s. Poss(a, s) \rightarrow \bigvee_{i=1}^{n} Step_i(a, s) \wedge Poss_i(a, s)$;
- $\forall s. Final(s) \rightarrow \bigwedge_{i=1}^{n} Final_i(s)$.

Observe that, due to the last two axioms, the resulting theory Γ_C is not a Basic Action Theory. In Γ_C, we do not have anymore complete knowledge on the value of the fluents of the various *e*-Services. This is due to the new form of the successor-state axioms, which make fluents depend on the predicates $Step_i$, whose value is not determined uniquely by Γ_C. Note however that if we did know such values in every situation, then the value of all the fluents would be determined. Note also that the value of $Step_i$ is constrained by the last two axioms so that, in every situation that is not final for the target *e*-Service E_0, at least one of the component *e*-Services steps forward. Finally, the last axiom states that, if E_0 is in a final situation, then so are all component *e*-Services.

It can be shown that Γ_C *(i)* characterizes all the internal execution trees that conform to the external execution tree generated by Γ_0; *(ii)* delegates all actions to $E_1, \ldots, E_n$; *(iii)* is coherent with $E_1, \ldots, E_n$. More precisely it can be shown that from each model of Γ_C one can construct one such internal execution tree and that on the other hand starting from each such internal execution tree one can construct a model of Γ_C.

This characterization allow us to reduce checking for the existence of a composition to checking satisfiability of a propositional Situation Calculus theory.

Theorem 4. *Let $\Gamma_0, \Gamma_1, \ldots, \Gamma_n$ be the Basic Action Theories representing the* e*-Services $E_0, E_1, \ldots, E_n$ respectively, and let Γ_C be the theory defined as above. Then, Γ_C is satisfiable if and only if E_0 can be composed using $E_1, \ldots, E_n$.*

10.1 DPDL encoding

Once we have characterized *e*-Services and the problem of *e*-Service composition in Situation Calculus, we address the synthesis of a composite *e*-Service. We propose two equivalent approaches, based on reductions to checking satisfiability of a DPDL formula and of a Description Logics knowledge base, respectively. Indeed, we know from [16] that DLs and PDLs are equivalent. In the next subsection, we discuss the DL-based approach; in what follows we consider the DPDL-based one.

Specifically, we define a mapping δ from (uniform) Situation Calculus formulas with a free situation variable s to propositional DPDL formulas as follows:

$$\begin{aligned}
\delta(F(s)) &= F, \quad \text{for each fluent } F \\
\delta(Poss(a,s)) &= Poss_a, \quad (\text{sim. for } Poss_i(a,s)) \\
\delta Step_i(a,s) &= Step_a_i, \quad \text{for each } i \in 1..n \\
\delta(\neg\varphi(s)) &= \neg\delta(\varphi(s)) \\
\delta(\varphi_1(s) \wedge \varphi_2(s)) &= \delta(\varphi_1(s)) \wedge \delta(\varphi_2(s))
\end{aligned}$$

Next, we define the DPDL counterpart Δ_C of Γ_C as the conjunction of the following formulas.

- to model the situation tree, we add the conjunct $[u](\bigwedge_{a\in\Sigma}\langle a\rangle\texttt{true})$, and implicitly take into account the tree model property;
- to model the initial situation S_0, we add the conjunct $\delta(S_0)$; [9]
- for each precondition axiom $\forall s.Poss(a,s) \equiv \Psi_a(s)$, we add the conjunct $[u](\delta(Poss(a,s)) \equiv \delta(\Psi_a(s)))$; similarly for the modified precondition axioms in $\Gamma'_1, \ldots, \Gamma'_n$;
- for each successor-state axiom $\forall a, s.F(do(a,s)) \equiv \Phi_F(a,s)$, we first instantiate the axiom for each action in Σ and we simplify the equalities on actions. Then, for each instantiated successor-state axiom $F(do(\bar{a},s)) \equiv \Phi^{\bar{a}}_F(s)$ – where $\Phi^{\bar{a}}_F(s)$ is what we obtain from $\Phi_F(a,s)$ once we instantiate it on the action $\bar{a}$ and resolve the equalities on actions – we add the conjunct $[u]([\bar{a}]F \equiv \delta(\Phi^{\bar{a}}_F(s)))$;
- for the last two axioms of Γ_C, we add the conjuncts $[u](Poss\_a \wedge Final \rightarrow \bigvee_{i=1}^{n} Step\_a_i \wedge Poss\_a_i)$ and $[u](Final \rightarrow \bigwedge_{i=1}^{n} Final_i)$.

Note that, in the above construction, it is necessary to instantiate the successor-state axioms for each action, since, contrary to the Situation Calculus, DPDL does not admit quantification over actions.

Theorem 5. *The DPDL counterpart Δ_C of Γ_C is satisfiable if and only if Γ_C is so.*

Proof (sketch). Given a model of Γ_C, one can easily construct a model of Δ_C. For the converse, we need to resort to the tree model property, and show that for every tree model of Δ_C (possibly obtained by unwinding an arbitrary model), we get a model of Γ_C. □

Observe that the size of Δ_C is at most equal to the size of Γ_C times the number of actions in Σ. Hence, from the EXPTIME-completeness of satisfiability in DPDL and from Theorem 5 we get again the following complexity result.

Theorem 6. *Checking the existence of an* e*-Service composition can be done in EXPTIME.*

Observe that, because of the small model property, from Δ_C one can always obtain a model which is at most exponential. From such a model one can immediately extract a finite (possibly exponential) representation of the composition labeling.

[9] Note that $[u]\cdot$ does not appear in front of the propositional formula S_0.

10.2 DL encoding

We show how to synthesize a composite e-Service by re-expressing Situation Calculus Action Theories as an $\mathcal{ALCU}$ knowledge base, a well known Description Logic [4]. $\mathcal{ALCU}$ concepts are built by starting from atomic concepts and atomic roles as follows:

$$C \longrightarrow A \mid \neg A \mid C_1 \sqcap C_2 \mid C_1 \sqcup C_2 \mid \forall R.C \mid \exists R.\top$$

where A is an atomic concept and R is an atomic role. An $\mathcal{ALCU}$ knowledge base is a set of inclusion assertions of the form

$$C_1 \sqsubseteq C_2$$

where C_1, C_2 are arbitrary $\mathcal{ALCU}$ concepts. We also use the abbreviation $C_1 \equiv C_2$ for $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$. As for reasoning service we concentrate on concept satisfiability in a knowledge base, which is easily shown to be EXPTIME-complete for $\mathcal{ALCU}$, since concept satisfiability in a knowledge base is already EXPTIME-hard for $\mathcal{AL}$ and is EXPTIME-complete for $\mathcal{ALC}$ which includes $\mathcal{ALCU}$ (see [4] for details).

$\mathcal{ALCU}$ (as well as $\mathcal{ALC}$) enjoys three properties that are of particular interest for our aims. The first is the *tree model property*, which says that every model of a concept in a knowledge base can be unwound to a (possibly infinite) tree. The second is the *small model property*, which says that every satisfiable concept in a knowledge base admits a finite model of size at most exponential in the size of the concept and the knowledge base itself. The third is the *single successor property* that says that every model of a concept in a knowledge base can be transformed in such a way that in each object there is at most a unique R-successor for each role R. Moreover such a transformation does not increase the size of the model.

Similarly as before, we define a mapping δ from (uniform) Situation Calculus formulas (wlog in negation normal form) with a free situation variable s to boolean combination of concepts as follows:

$$\begin{aligned}
\delta(F(s)) &= F, \quad \text{for each fluent } F \\
\delta(Poss(a,s)) &= Poss_a, \quad (\text{similarly for } Poss_i(a,s)) \\
\delta(Step_i(a,s)) &= Step_a_i, \quad \text{for each } i \in 1..n \\
\delta(\neg\varphi(s)) &= \neg\delta(\varphi(s)) \quad (\varphi \text{ is an atomic proposition}) \\
\delta(\varphi_1(s) \wedge \varphi_2(s)) &= \delta(\varphi_1(s)) \sqcap \delta(\varphi_2(s)) \\
\delta(\varphi_1(s) \vee \varphi_2(s)) &= \delta(\varphi_1(s)) \sqcup \delta(\varphi_2(s))
\end{aligned}$$

Also, we consider an $\mathcal{ALCU}$ role for each atomic action in Σ.

Next, we define the $\mathcal{ALCU}$ counterpart Δ_C of Γ_C as the following knowledge base.

- to model the situation tree, we add the assertion $\top \sqsubseteq \bigsqcap_{a\in\Sigma} \exists a.\top$, and implicitly take into account the tree model property and the unique successor property;
- to model the initial situation $\mathcal{S}_0$, we add the assertion $Init \sqsubseteq \delta(S_0)$, where $Init$ is a new atomic concept denoting the initial situation;
- for each precondition axiom $\forall s.Poss(a,s) \equiv \Psi_a(s)$, we add the assertion $\delta(Poss(a,s)) \equiv \delta(\Psi_a(s))$; similarly for the modified precondition axioms in $\Gamma'_1, \ldots, \Gamma'_n$;
- for each successor-state axiom $\forall a, s.F(do(a,s)) \equiv \Phi_F(a,s)$, we first instantiate the axiom for each action in Σ and we simplify the equalities on actions. Then, for each instantiated successor-state axiom $F(do(\bar{a},s)) \equiv \Phi^{\bar{a}}_F(s)$ – where $\Phi^{\bar{a}}_F(s)$ is what we obtain from $\Phi_F(a,s)$ once we instantiate it on the action $\bar{a}$ and resolve the equalities on actions – we add the assertion $\forall \bar{a}.F \equiv \delta(\Phi^{\bar{a}}_F(s))$;

– for the last two axioms of Γ_C, we add the assertions $Poss\_a \sqsubseteq \bigsqcup_{i=1}^{n} Step\_a_i \sqcap Poss\_a_i$ and $Final \sqsubseteq \bigsqcap_{i=1}^{n} Final_i$.

Observe that, in the above construction, it is necessary to instantiate the successor-state axioms for each action, since, contrary to the Situation Calculus, $\mathcal{ALU}$ does not admit quantification over actions.

Theorem 7. *The Init concept is satisfiable in the $\mathcal{ALU}$-counterpart Δ_C of Γ_C if and only if Γ_C is satisfiable.*

Observe that the size of Δ_C is at most equal to the size of Γ_C times the number of actions in Σ. Hence, from the EXPTIME-completeness of concept satisfiability in $\mathcal{ALU}$ knowledge bases and from Theorem 7 we get again the following complexity result.

Theorem 8. *Checking the existence of an* e-*Service composition can be done in EXPTIME.*

Observe that, because of the small model property and the single successor property, if *Init* is indeed satisfiable in Δ_C one can always obtain a model which is single successor and of size at most exponential. From such a model one can immediately extract a finite (possibly exponential) representation of the internal execution tree constituting the composition. Also from such a representation one can build a Situation Calculus Basic Action Theory (or its counterpart in $\mathcal{ALU}$ if needed) that generates exactly such a internal execution tree.

Note that all approaches lead to the same lower bound as far complexity of composition, as Theorems 2, 6 and 8 show.

11 Future Work

The main contribution of our 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. For example, it is interesting to give a lower bound characterization for the complexity of the composition problem. Additionally, we can take into account, in our setting, the presence of incomplete information [7], as well as of communication delays and of an unbounded number of active instances. We are also developing a DL-based prototype system that implements the devised composition technique.

However, from our research on *e*-Services, even more far-reaching open issues may be identified. We may define a real-business scenario for finite state *e*-Service composition, where *e*-Services are expressed in terms of standard formalisms such as those based on UML statecharts. Then, we can invoke the composition system under development on such *e*-Services and test how the complexity of composition in our framework impacts the real world applications. Another possible research line may concern studying if and how our mediated approach can evolve towards a peer-to-peer one. Indeed, in our framework, only after the composite *e*-Service has been synthesized (off-line), the orchestrator invokes each component *e*-Service accordingly. Therefore, for each action to be performed by the composite *e*-Service, the orchestrator calls the component *e*-Service that executes it; in particular, the orchestrator may repeatedly call the same *e*-Service to have it execute a sequence of actions. In such a case, instead, it would be desirable to identify "bulks" of consecutive

actions executed by a same *e*-Service, and to call such an *e*-Service just once. Taking this situation to the extreme, it is possible to identify situations when the composite *e*-Service can be decomposed into simpler, concurrent finite state machines, to be executed by the component *e*-Services that exchange synchronization messages among them, in order to decide when which *e*-Service executes which action(s). In such a case the orchestrator is dropped out, and the user directly interacts with the component *e*-Services in the community. This implies that the component *e*-Services become more complex since they have to implement synchronization logic and communication delays. Of course, it is interesting also to study scenarios depending on the number of synchronization messages exchanged. As an example, when no synchronization message is present, the user interacts with only one component *e*-Service.

The research in *e*-Service composition will also be performed within two projects, currently under review. The first one, a COFIN project, aims at studying and applying *e*-Services and *e*-Service composition to a domotic context[10]. Our Department participates to this project as the national coordinator. The second project is an european Network of Excellence aiming at defining a general framework for *e*-Services from a point of view which is both theoretical and application oriented. Our Department participates to this network as the coordinator of the work-package about the definition of a theoretical framework for *e*-Service composition.

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. Ariba, Microsoft, and IBM. Web Services Description Language (WSDL) 1.1. W3C Note. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`, March 2001.
4. 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.
5. BEA, Intalio, SAP, and Sun. Web Service Choreography Interface (WSCI) 1.0. W3C Document. `http://www.w3.org/TR/wsci/`, 2002.
6. Mordechai Ben-Ari, Joseph Y. Halpern, and Amir Pnueli. Deterministic Propositional Dynamic Logic: Finite Models, Complexity, and Completeness. *J. of Computer and System Sciences*, 25:402–417, 1982.
7. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. *e*-Service Composition by Description Logic Based Reasoning. In *Proceedings of the Int. Workshop on Description Logics (DL03)*, Rome, Italy, 2003.
8. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of *e*-Services that Export their Behavior. In *Proceedings of the First International Conference on Service Oriented Computing (IC-SOC03)*, Trento, Italy, 2003.
9. 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.
10. D. Berardi, D. Calvanese, G De Giacomo, and M. Mecella. Composing e-Services by Reasoning about Actions. In *Proc. of the ICAPS 2003 Workshop on Planning for Web Services*, 2003.

[10] Domotics is a multidisciplinary field where concepts from pervasive computing, networking, automation, artificial intelligence, web services, robotics (to cite a few of them) play their role in order to make our home a more comfortable place to live, especially, for the elderly and disabled people. Hence, the key idea behind the "domotic home" is to enable the automation of human tasks as annoying/repetitive as common in-home activities use to be. To obtain such results, domotic researchers envision a house environment populated by several advanced "services" (household electronic appliances, personal computers, personal digital assistants, remote controllers, personal robots, ...) working together to push the quality and the scope of the service to the end-user (the human inhabitant) far beyond the limit of standard house facilities.

11. 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.
12. 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.
13. 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.
14. F. Casati and M.C. Shan. Dynamic and Adaptive Composition of *e*-Services. *Information Systems*, 6(3), 2001.
15. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services (Version 1.0). IBM Document. `http://www.ibm.com/developerworks/library/ws-bpel/`, July 2002.
16. Giuseppe De Giacomo and Maurizio Lenzerini. Boosting the Correspondence between Description Logics and Propositional Dynamic Logics. In *Proc. of the 12th Nat. Conf. on Artificial Intelligence (AAAI'94)*, pages 205–212, 1994.
17. Giuseppe De Giacomo and Maurizio Lenzerini. PDL-Based Framework for Reasoning about Actions. In *Proc. of the 4th Conf. of the Ital. Assoc. for Artificial Intelligence (AI*IA'95)*, volume 992 of *Lecture Notes in Artificial Intelligence*, pages 103–114. Springer, 1995.
18. ebXML.org. Business Process Specification Schema (Version 1.01). `http://www.ebXML.org/specs/ebBPSS.pdf`.
19. ebXML.org. Collaboration-Protocol Profile and Agreement Specification (Version 1.0). `http://www.ebXML.org/specs/ebCCP.pdf`.
20. ebXML.org. Registry Information Model (Version 1.0). `http://www.ebXML.org/specs/ebRIM.pdf`.
21. 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.
22. Michael J. Fischer and Richard E. Ladner. Propositional Dynamic Logic of Regular Programs. *J. of Computer and System Sciences*, 18:194–211, 1979.
23. Günter Gans, Matthias Jarke, Gerhard Lakemeyer, and Dominik Schmitz. Deliberation in a Modeling and Simulation Environment for Inter-organizational Networks. In *Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE'03)*, Velden, Austria, 2003.
24. Volker Haarslev and Ralf Möller. Description of the RACER System and its Applications. In *Proc. of the 2001 Description Logic Workshop (DL 2001)*, pages 132–141. CEUR Electronic Workshop Proceedings, `http://ceur-ws.org/Vol-49/`, 2001.
25. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. The MIT Press, 2000.
26. 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.
27. 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.
28. 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.
29. Carsten Lutz and Ulrike Sattler. A Proposal for Describing Services with DLs. In *Proc. of the 2002 Description Logic Workshop (DL 2002)*, pages 128–139. CEUR Electronic Workshop Proceedings, `http://ceur-ws.org/Vol-53/`, 2002.
30. John McCarthy and Patrick J. Hayes. Some Philosophical Problems from the Standpoint of Aritificial Intelligence. *Machine Intelligence*, 4:463–502, 1969.
31. 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.
32. 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.p%df`).
33. 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.
34. 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.
35. Mike P. Papazoglou. Agent-Oriented Technology in Support of e-Business. *Communications of the ACM*, October 2003. To appear.

36. 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.
37. Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems.* The MIT Press, 2001.
38. 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.
39. G. Shegalov, M. Gillmann, and G. Weikum. XML-enabled Workflow Management for *e*-Services across Heterogeneous Platforms. *Very Large Database J.*, 10(1), 2001.
40. SnehalThakkar, Craig Knoblock, and Jose Luis Ambite. A View Integration Approach to Dynamic Composition of Web Services. In *Proc. of the ICAPS 2003 Workshop on Planning for Web Services*, 2003.
41. 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.
42. 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.
43. Eric S.K. Yu. *Modelling Strategic Relationships for Process Reengineering.* PhD thesis, Dept. of Computer Science, University of Toronto, Toronto, ON, 1995.

A Deterministic Propositional Dynamic Logic

Propositional Dynamic Logics (PDLs) are a family of modal logics specifically developed for reasoning about computer programs [28]. 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 [25].

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[11]; $P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ 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:

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

[11] 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.

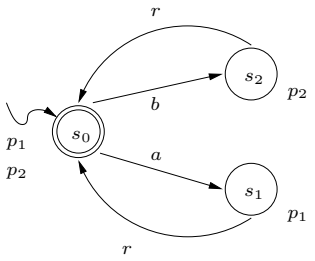


Fig. 6. Kripke structure for Example 4.

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:

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

Example 4. 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 6 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 [22].

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

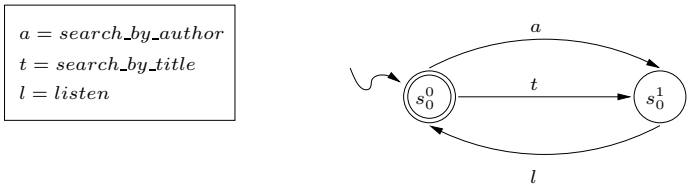
(considering domain elements as nodes and partial functions interpreting actions as edges). The second is the *small model property*, which says that every satisfiable formula admits a finite model whose size (in particular the number of domain elements) is at most exponential in the size of the formula 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 [6]:

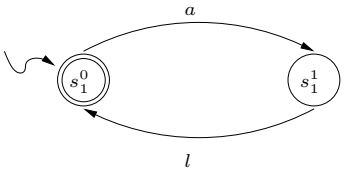
Theorem 9. *Satisfiability in DPDL is EXPTIME-complete.*

B Automatic *e*-Service Composition: an Example

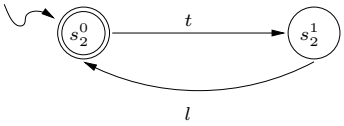
In order to make clearer the composition algorithm discussed in Section 9, we show how to build a MFSM for the target *e*-Service whose external schema is represented in Figure 7(a), and for the setting specified below.



(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. 7. Composition of *e*-Services

Figure 7(a) shows the external schema of the (target) *e*-Service E_0 of Example 1, specified by the client as a FSM A_0. Figure 7 (b) and (c) show the external schemas, 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 schemas 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.

In what follows, we first show how to build a possibly infinite model $\mathcal{I}$ for the DPDL formula Φ encoding A_0, A_1 and A_2, constituted as in Section 9. 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. Then, assuming to have derived from $\mathcal{I}$ a finite model $\mathcal{I}_f$ for Φ[12], 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 7 holds.

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

Specifically for our example, the set of atomic actions $\mathcal{A}$, the set of atomic programs $\mathcal{P}$ and Φ are defined as follows:

The set $\mathcal{P}$ of atomic propositions is:

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

where,

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

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

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

where:

- a denotes action `search_by_author`
- t denotes action `search_by_title`
- l denotes action `listen`

The master modality is:

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

i.e., 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 the target e-Service E_0 are:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Finally, the following formulas must hold for the overall composition.

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

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

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

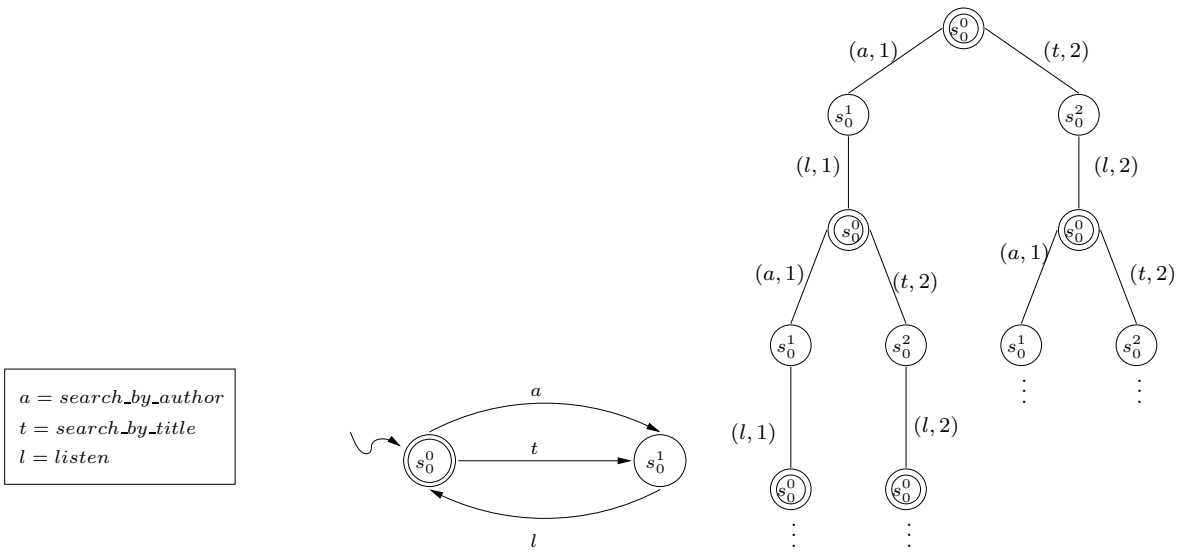
We assume that, given the component FSMs A_1 and A_2 there exists a composite e-Service having FSM A_0 as external schema and A_1 and A_2 as components. Let $T({E_0}^{int})$ be the internal execution tree for E_0 wrt the community C to which E_1 and E_2 belong, such that: *(i)* $T({E_0}^{int})$ conforms to $T(A_0)$, i.e., to the external execution tree obtained from A_0 as in Section 8, *(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.

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

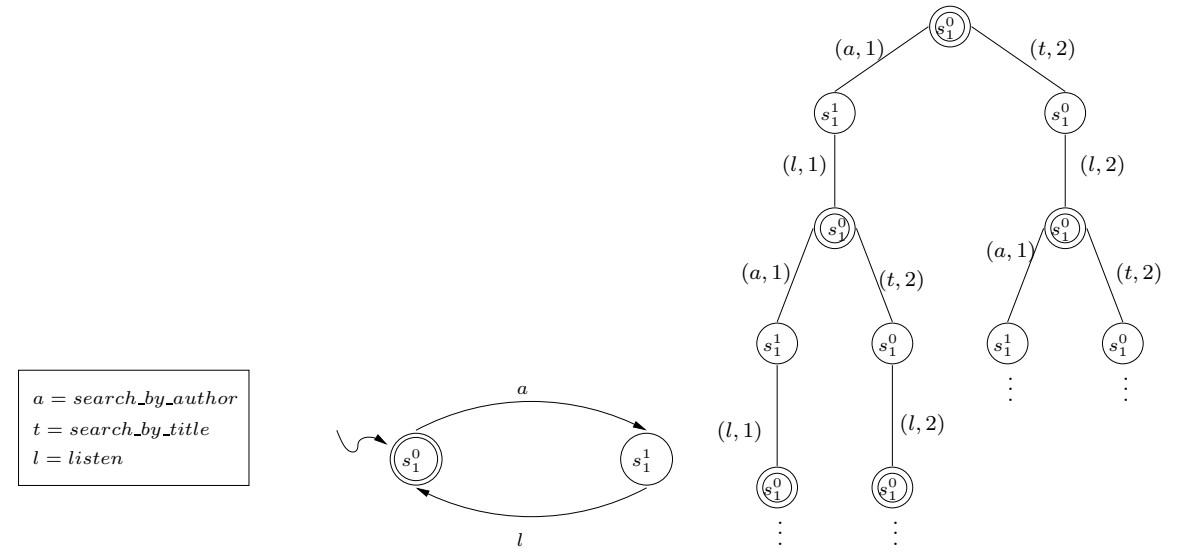
Figure 8(b) represents the internal execution tree of E_0, where each node is labeled with the corresponding state of the automaton. σ maps over s_0^1 the nodes of the tree that represent strings ending by a or t; it maps over s_0^0 the root of the tree and the nodes of the tree associated to strings ending by l.

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



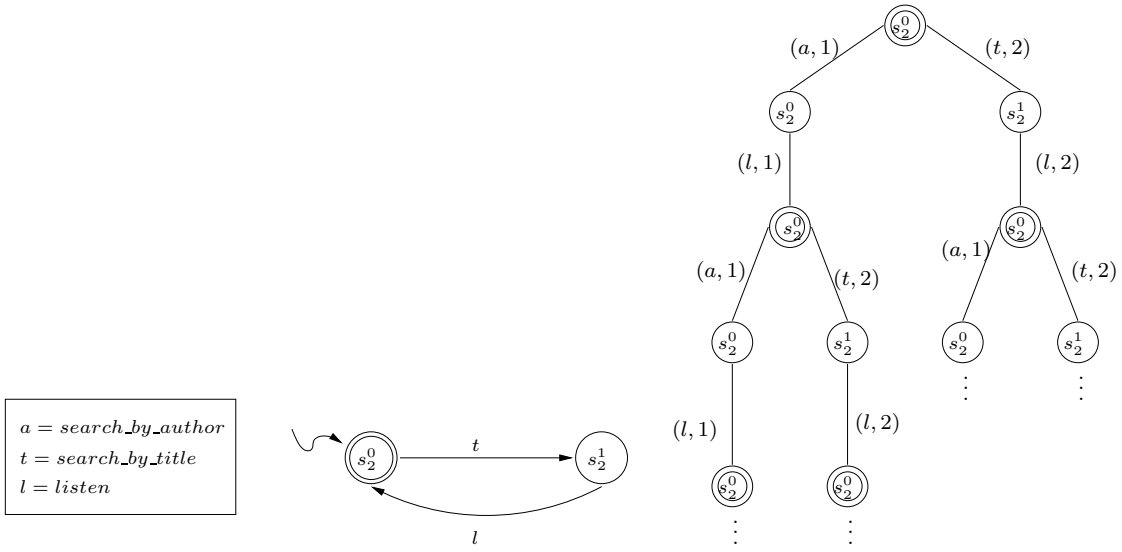
(a) External schema A_0 of target e-Service E_0 (b) The mapping σ on $T({E_0}^{int})$

Fig. 8. Composition of e-Services



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

Fig. 9. 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. 10. Composition of e-Services

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

Figure 9(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.

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

Figure 10(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

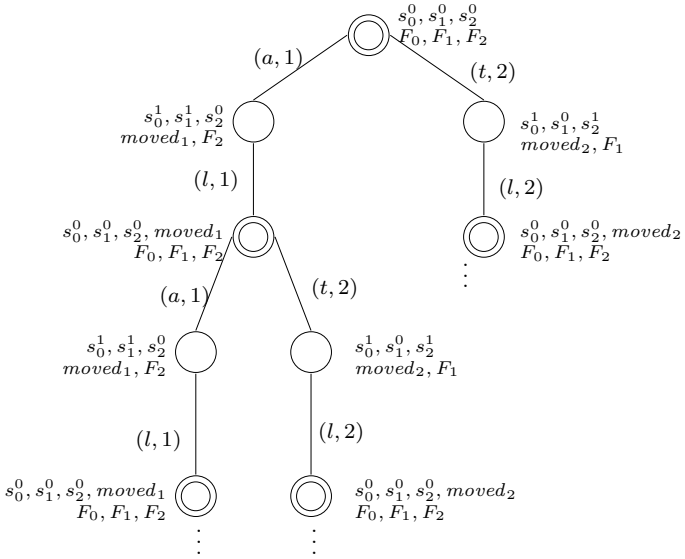


Fig. 11. Infinite model $\mathcal{I}$ for Φ.

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 11 shows that $\mathcal{I}$ is a model for the formula Φ[13]. 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 12 as a FSM.

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

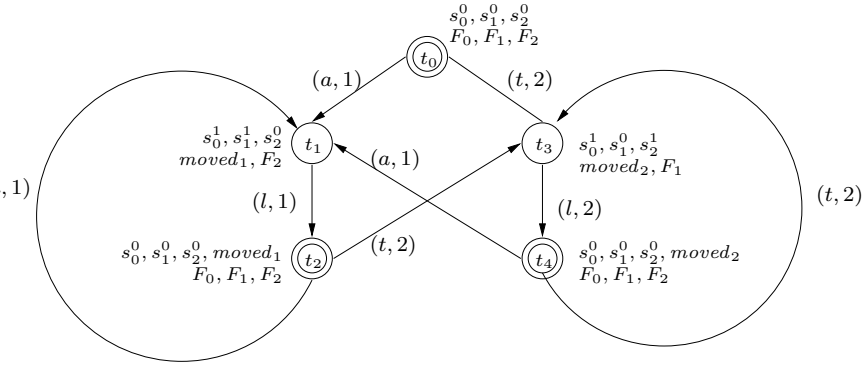


Fig. 12. Finite model $\mathcal{I}_f$ for Φ.

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.

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

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.

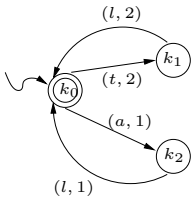


Fig. 13. Minimal FSM associated to $T({E_0}^{int})$.

– 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 FSM associated to the tree representing the infinite model is shown in Figure 13. It is easy to see that it does not represent a model for Φ since, for instance, state k_0, which corresponds to both states t_0 and t_1 in Figure 12, is associated to both $moved_1$ and $\neg moved_1$.