Reasoning about Actions for *e*-Service Composition

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

Abstract

Composition of *e*-Services is the issue of synthesizing a new *composite e*-Service, obtained by combining a *set* of available *component e*-Services, when a client request cannot be satisfied by available *e*-Services. In this paper we study the problem of composition synthesis in a general framework. We consider *e*-Services as arbitrary (possibly infinite) execution trees, i.e., as trees of all potential interactions with clients, and characterize composition in this abstract setting. We then show how this setting can be realized using Reasoning About Actions, in particular reasoning in Situation Calculus, and exploiting a correspondence with Deterministic Propositional Dynamic Logic, we provide automated procedures and complexity results for performing composition.

Introduction

e-Services, also referred to as Web Services, represent a new model in the utilization of the Web, in which self-contained, modular applications can be described, published, located and invoked dynamically over the Web, in a programming language independent way. The commonly accepted and *minimal* framework for *e*-Service, referred to as Service Oriented Architecture (SOA (Pilioura & Tsalgatidou 2001)), consists of some basic roles: *(i)* the service provider, *(ii)* the service directory and *(iii)* the service requestor.

The *service provider* is the subject (e.g., an organization) providing services; available services are described using a service description language, and advertised on a public available service directory. The *service directory* is the subject providing a repository/registry of service descriptions, where providers publish their services and requestors find services. The *service requestor*, also referred to as client, is the subject looking for and invoking the service in order to fulfill some goals; it can be either a human or another service. A requestor discovers the most suitable service in the directory, then it connects to the specific service provider and uses the service.

Composition of *e*-Services addresses the situation when a client request cannot be satisfied by an available *e*-Service, but a *composite e*-Service, obtained by combining a *set* 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, 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. The second, sometimes called *composition synthesis*, or simply *composition*, is concerned with automatically synthesizing a new *e*-Service starting from component *e*-Services, thus producing a specification of how to coordinate the component *e*-Services to obtain the composite *e*-Service.

Several recent works address the problem of orchestration. In (Casati & Shan 2001), an *e*-Service that performs coordination of *e*-Services is considered as a (meta) *e*-Service, referred to as Composite *e*-Service, that can be transparently invoked by clients. In (Fauvet *et al.* 2001), a composite *e*-Service is modeled as an activity diagram, and its enactment is carried out through the coordination of different state coordinators (one for each component *e*-Service and one for the composite *e*-Service itself), in a decentralized way, through peer-to-peer interactions. In (Shegalov, Gillmann, & Weikum 2001), coordination of *e*-Services is obtained by an enactment engine interpreting process schemas modeled as statecharts (Wodtke & Weikum 1997). Finally, in (Mecella, Parisi Presicce, & Pernici 2002), orchestration of *e*-Services is addressed by means of Petri Nets.

Instead, composition synthesis remains still almost unexplored. Interestingly, the best known contributions are based on planning and reasoning about actions in AI. In particular, in (Aiello *et al.* 2002) a way of composing *e*-Services is presented, based on planning under uncertainty and constraint satisfaction techniques, and a request language for specifying client goals is proposed. In (McIlraith, Son, & Zeng 2001; McIlraith & Son 2002), composition of *e*-Services is addressed by using the Situation Calculus-based programming language CONGOLOG; specifically, component *e*-Services are represented as CONGOLOG programs, while the client's needs are specified through suitable forms of constraints. Composition synthesis is based on using such constraints for pruning the tree of possible executions of the available e-Services.

In this paper, we study the problem of composition synthesis in a general framework. We consider e-Services as arbitrary (possibly infinite) execution trees, i.e., as trees of all potential interactions with clients. An interaction consists of the client invoking a command and waiting for the fulfillment of the task and the return of some information. In this setting, we define what a composition of the component e-Services in order to realize a target e-Service amounts to. Namely, it consist in deciding for each interaction with the client in the target e-Service, which of the component e-Services should execute it, in such a way that each component e-Service executes computations that are legal according to its execution tree. Then we show that this abstract model of composition synthesis can in fact be realized using reasoning about actions formalisms to represent the execution trees of the e-Services, and satisfiability for synthesizing the composition. In particular, we focus on one of the best known of such formalisms: the propositional variant of Reiter's Situation Calculus Basic Action Theories. Finally, we show that we can use reasoning procedures developed for Propositional Dynamic Logics (Kozen & Tiuryn 1990) to perform compositions in this case, thus getting effective algorithms and complexity results.

The remainder of this paper is as follows. Section presents the formal framework for representing e-Services and e-Service composition based on execution trees. Section shows how we can use reasoning about actions formalisms, and in particular Situation Calculus, for representing e-Services and synthesizing compositions. Section shows the use of Propositional Dynamic Logics reasoning procedures for getting effective algorithms and complexity results to perform composition synthesis. Finally, Section concludes the paper.

Framework

In this work, an e-Service is considered as a "black-box" software artifact (delivered over the Internet) that executes certain programs interacting with its clients (either human users or other e-Services). An *interaction* consists of a client invoking a *command* on the e-Service and waiting for the fulfillment of the specific task and (possibly) the return of some information.

After executing the computation triggered by the invoked command, the e-Service is ready to receive new commands. In general, not all commands are invocable at a given point: the possibility of invoking them depends on the previously executed ones. On the basis of the information returned by the previous commands, the client chooses the next interaction to perform.

Under certain circumstances the e-Service can be considered terminated and the client can stop stepping through it. Notice that, in principle, it may be that a given e-Service needs to interact with the client for an unbounded, or even infinite, number of steps, thus providing to the client a continuous service.

When a client needs a new e-Service, he asks to an e-*Service Integration System* to synthesize a *composite* e-*Service* , i.e., a new e-Service obtained by coordination of different available e-Services, in order to fulfill the client's needs.

From the specification of the client e-Service and the available e-Services, the e-Service Integration System derives a specification of how to orchestrate the available e-Services such that the e-Service requested by the client is realized. The e-Service Integration System keeps the formal specifications of the e-Services registered into the service directory. The *composer* module, which is the core of the system, specifically addresses the issue of determining how to compose the available e-Services. After being synthesized, composite e-Services are made available for execution. Clients, during interactions, are not aware they are interacting with a composite e-Service instead of simpler ones; all happens in a transparent manner for clients. Composite e-Services are not executed directly by e-Service providers, but their specification is enacted by the e-Service Integration System by suitably orchestrating the available e-Services. We will not address the issues of execution further, since they go beyond the scope of our paper. Note that in this framework, we have made the assumption that the e-Service Integration System has complete knowledge on the available e-Services and the e-Service requested by the client.

In this paper, e-Services will be considered from an abstract point of view. In particular, each e-Service is represented as an *execution tree*, i.e., as the tree of all potential interactions of the e-Service with the client. We do not represent the information returned to the client, since the purpose of such information is to let the client choose the next command, and the rationale behind this choice depends entirely on the client.

As usual, we consider a tree $\mathcal{T}$ over an alphabet Σ as a prefix closed (possibly infinite) set of finite words over Σ, i.e., a set of words $\mathcal{T} \subseteq \Sigma^*$, called *nodes*, such that if $x \cdot c \in \mathcal{T}$, with $x \in \Sigma^*$ and $c \in \Sigma$, then also $x \in \mathcal{T}$. The empty word ε is called the *root* of $\mathcal{T}$, and for every $x \in \mathcal{T}$, the node $x \cdot c$, with $c \in \Sigma$, is called the *successor* of x. A *labeled tree* is a pair $(\mathcal{T}, f)$, where $\mathcal{T}$ is a tree and f is a *labeling function* assigning to each node of $\mathcal{T}$ an element of a given labeling domain.

Let $e\mathcal{S}$ be an e-Service, and let Σ be the alphabet of interactions supported by the e-Service. The *execution tree* of $e\mathcal{S}$ is a *labeled tree* $\mathcal{T}^{e\mathcal{S}} = (\mathcal{T}, \mathit{fin})$, where $\mathcal{T}$ is a tree over Σ and *fin* is a boolean labeling function. Intuitively:

- Each node of $\mathcal{T}^{e\mathcal{S}}$ represents the history of the executed sequence of interactions between the client and the e-Service.
- The root of $\mathcal{T}^{e\mathcal{S}}$ represents the fact that the client has not yet performed any interaction with the e-Service.
- For every node x in $\mathcal{T}^{e\mathcal{S}}$, the successor $x \cdot c$ represents the fact that, after performing the sequence of interactions x, the client chooses to invoke the command c, among those available. Therefore, each node represents a choice point at which the client makes a decision on the next (atomic) computation the e-Service should perform.
- The nodes that are labeled `true` by *fin* are called *final*, and represent the fact that the e-Service can successfully

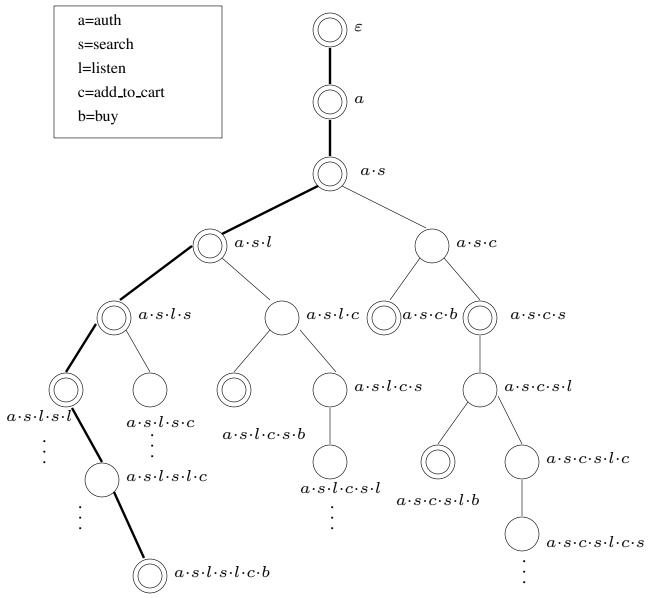

Figure 1: Example of *e*-Service execution tree

terminate its computation at this point. In other words, the client can quit an *e*-Service when it is in a final node, and only then.[1]

Example 1 Figure 1 shows an execution tree representing an *e*-Service that allows for searching and buying mp3 files[2]. After an authentication step (interaction `auth`), in which the client provides *userID* and *password*, the *e*-Service waits for searching parameters (e.g., author or group name, album or song title) and returns a list of matching files (interaction `search`); then, the client can: *(i)* select and listen to a song (interaction `listen`), and choose if performing another `search` or if adding the selected file to the cart (interaction `add_to_cart`); *(ii)* `add_to_cart` a file without listening to it. Then, the client chooses if performing those interactions again. Finally, by providing its payment method details the client buys and downloads the contents of the cart (interaction `buy`).

In the figure, a specific sequence of interactions is highlighted (shown in thick lines); it corresponds to an *e*-Service execution in which the client provides its credentials and search parameters, searches for and listens to two mp3 files, adds the latter to the cart, and buys it.

Note that, after the interaction `auth`, the client may quit the *e*-Service since he may have submitted wrong authentication parameters. On the contrary, the client is forced to buy, within the single interaction `buy`, a certain number of selected songs, contained in the cart, possibly after choosing and listening some songs zero or more times. ■

We are now ready to give the definition of *e*-Service composition. Let $e\mathcal{S}_1, \ldots, e\mathcal{S}_n$ be n *e*-Services, called *component* e-*Services* , defined over the same set of interactions Σ, and let $e\mathcal{S}_0$ be the *e*-Service, called *target* e-*Service* , requested by the client (also defined over Σ). Let $\mathcal{T}_i^{e\mathcal{S}} = (\mathcal{T}_i, \mathit{fin}_i)$ be the execution tree of $e\mathcal{S}_i$, for $i = 0, 1, \ldots, n$. We call *composition labeling* of $\mathcal{T}_0^{e\mathcal{S}}$ wrt $\mathcal{T}_1^{e\mathcal{S}}, \ldots, \mathcal{T}_n^{e\mathcal{S}}$ a labeling function $comp : \mathcal{T}_0 \to \mathcal{T}_1 \times \cdots \times \mathcal{T}_n$ that satisfies the following conditions:

1. $comp(\varepsilon) = \langle \varepsilon, \ldots, \varepsilon \rangle$;
2. for every node $x \in \mathcal{T}_0$, let $comp(x) = \langle x_1, \ldots, x_n \rangle$; then $comp(x{\cdot}c) = \langle y_1, \ldots, y_n \rangle$, where $y_i = x_i{\cdot}c$ if the *e*-Service $e\mathcal{S}_i$ performs the interaction c, and $y_i = x_i$ otherwise; moreover, $y_i = x_i{\cdot}c$ for at least one i.
3. for every node $x \in \mathcal{T}_0$, if $\mathit{fin}_0(x) = \texttt{true}$ and $comp(x) = \langle x_1, \ldots, x_n \rangle$, then $\mathit{fin}_i(x_i) = \texttt{true}$ for $i = 1, \ldots, n$.

Intuitively, $comp$ labels each node of $\mathcal{T}_0^{e\mathcal{S}}$ by an n-tuple $\langle x_1, \ldots, x_n \rangle$, where each x_i denotes the current node of the execution tree $\mathcal{T}_i^{e\mathcal{S}}$. Requirement (1) on the root states that all *e*-Services start from the beginning of their computation. Requirement (2) models that each interaction of the target *e*-Service is in fact executed by at least one, but possibly many, of the component *e*-Services. Requirement (3) models the fact that, when the target *e*-Service can be left, then all component *e*-Services must be in a final configuration so that they can terminate.

We say that a target *e*-Service $e\mathcal{S}_0$ can be *composed* using a set of component *e*-Services $e\mathcal{S}_1, \ldots, e\mathcal{S}_n$ if there exists a composition labeling $comp$ of $\mathcal{T}_0^{e\mathcal{S}}$ wrt $\mathcal{T}_1^{e\mathcal{S}}, \ldots, \mathcal{T}_n^{e\mathcal{S}}$. Indeed, if such a composition labeling exists, then one can orchestrate the n component *e*-Services to obtain $e\mathcal{S}_0$ by stepping each component *e*-Service according to what specified by the labeling $comp$.

e-Service Composition in Situation Calculus

We have characterized *e*-Service behavior and composition in general terms by means of execution trees. This abstract view needs to be refined in order to get a finite representation of *e*-Services that can be concretely manipulated.[3] In this paper we propose to use formalisms developed for Reasoning about Actions to represent *e*-Services, and to use logical reasoning, in particular, satisfiability, to solve the problem of *e*-Service composition. This approach gives us the ability of dealing with a large class of *e*-Services using a compact and high-level representation. There are many possible action languages that can be used for representing *e*-Services. Here we focus on Reiter's Situation Calculus Basic Action Theories (Reiter 2001), which are widely known and allow us to concentrate on the aspects specific to our problem. For the sake of simplicity, we consider them in a propositional setting.

We will not go over the Situation Calculus (McCarthy & Hayes 1969) 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,

[1]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.

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

[3]Obviously, not all execution trees can be represented in a finite way.

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 (Reiter 2001):

- 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 and in which $Poss$ does not appear. 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.

We represent each *e*-Service $e\mathcal{S}$ as a Basic Action Theory Γ, where each interaction 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. 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 to the *e*-Service Integration System, since they are not attached in any way to the actual *e*-Services. In contrast, actions represents interactions meaningful both to the client and the *e*-Service the client interacts with.

Intuitively, the part of the situation tree (Reiter 2001) formed only by the actions that are possible (as specified by $Poss$) directly corresponds to the execution tree 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 (i.e., nodes of the execution tree) 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* $\mathcal{T}^{e\mathcal{S}} = (\mathcal{T}, fin)$ *generated by* Γ is defined as $\mathcal{T} = \{n(s) \mid n(s) \neq undef\}$, and $fin(n(s)) = \texttt{true}$ iff $Final(s)$ holds. It is easy to check that $\mathcal{T}^{e\mathcal{S}}$ is indeed an execution tree.

Once we have settled how to represent *e*-Services in Situation Calculus, we turn to solving *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. The basic idea to model *e*-Service composition 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 *e*-Service $e\mathcal{S}_i$ 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: $\forall a, s.F_i(do(a, s)) \equiv (Step_i(a, s) \wedge \Phi_{F_i}(a, s)) \vee (\neg Step_i(a, s) \wedge F_i(s))$;
- $\forall a, s.(Poss(a, s) \wedge \neg Final(s)) \supset \bigvee_{i=1}^{n} Step_i(a, s) \wedge Poss_i(a, s)$
- $\forall s.Final(s) \supset \bigwedge_{i=1}^{n} Final_i(s)$

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 $e\mathcal{S}_0$, at least one of the component *e*-Services steps forward. Finally, the last axiom states that, if $e\mathcal{S}_0$ is final, then so are all component *e*-Services.

Theorem 1 *Let $\Gamma_0, \Gamma_1, \ldots, \Gamma_n$ be the Basic Action Theories representing the* e-*Services $e\mathcal{S}_0, e\mathcal{S}_1, \ldots, e\mathcal{S}_n$ respectively, and let Γ_C be the theory defined as above. Then, Γ_C is satisfiable if and only if $e\mathcal{S}_0$ can be composed using $e\mathcal{S}_1, \ldots, e\mathcal{S}_n$.*

Proof (sketch). Let $\mathcal{T}_0^{e\mathcal{S}}, \mathcal{T}_1^{e\mathcal{S}}, \ldots, \mathcal{T}_n^{e\mathcal{S}}$ be the execution trees generated by $\Gamma_0, \Gamma_1, \ldots, \Gamma_n$, respectively.

"If": If $e\mathcal{S}_0$ can be composed using $e\mathcal{S}_1, \ldots, e\mathcal{S}_n$, there exists a composition labeling $comp$ of $\mathcal{T}_0^{e\mathcal{S}}$ wrt $\mathcal{T}_1^{e\mathcal{S}}, \ldots, \mathcal{T}_n^{e\mathcal{S}}$. From $\mathcal{T}_0^{e\mathcal{S}}$ and $comp$ we can obtain a model $\mathcal{M}$ of Γ_C as follows.

First, from $\mathcal{T}_0^{e\mathcal{S}}$ it is straightforward to build a model $\mathcal{M}_0$ of Γ_0. From $comp$, we can then extend such a model to $\mathcal{M}'_0$ by adding the truth-value of $Step_i(a, s)$ for every component *e*-Service $e\mathcal{S}_i$, every action a, and every situation s. Now consider that $Poss_i$ and all fluents F_i, including $Final_i$, are completely determined once each $Step_i$ is determined, due to the form of precondition and successor-state axioms, respectively. This means that we can further extend $\mathcal{M}'_0$ to a model $\mathcal{M}$ of $\Gamma'_1, \ldots, \Gamma'_n$. Moreover, by construction of $comp$, such a model $\mathcal{M}$ satisfies the last two axioms of Γ_C, and hence the whole Γ_C.

"Only If": Let $\mathcal{M}$ be a model of Γ_C. Note that Γ_C is an extension of Γ_0, hence the execution tree generated by Γ_C is $\mathcal{T}_0^{e\mathcal{S}}$. Now, by using the truth values of $Step_i(a, s)$ for every component e-Service $e\mathcal{S}_i$, every action a, and every situation s, one can construct a labeling function $comp : \mathcal{T}_0^{e\mathcal{S}} \rightarrow \mathcal{T}_1^{e\mathcal{S}} \times \cdots \times \mathcal{T}_n^{e\mathcal{S}}$. Due to the constraints posed to the interpretation of $Step_1, \ldots, Step_n$ by the theory Γ_C, *comp* is indeed a composition labeling. $\Box$

Example 2 Suppose we have two *e*-Services, $e\mathcal{S}_1$ and $e\mathcal{S}_2$, both of them dealing with mp3 files. Specifically, $e\mathcal{S}_1$, allows a client to search a file on the basis of some searching parameters (e.g., author or group name, album or song title – interaction `search`) and then to select and listen to a song (interaction `listen`); this can be executed repeatedly and the *e*-Service can be quit at any time.

The *e*-Service $e\mathcal{S}_2$, consists of an authentication step (interaction `auth`), in which clients provide *userID* and *password*; then the *e*-Service waits for searching parameters and returns a list of matching files (interaction `search`); at this point the client can: *(i)* select and listen to a *sample* of a song (interaction `listen_sample`), and choose if performing another `search` or if adding the selected file to the cart (interaction `add_to_cart`); *(ii)* `add_to_cart` a file without listening to it. Then, the client chooses if performing those interactions again. Finally, by providing its payment method details the client buys and downloads the contents of the cart (interaction `buy`).

Note that the *e*-Service $e\mathcal{S}_2$ differs from the one proposed in Example 1 due to the interaction `listen_sample` instead of `listen`. Note also that the interaction `listen` is provided by $e\mathcal{S}_1$.

Under the assumption that the two *e*-Services $e\mathcal{S}_1$ and $e\mathcal{S}_2$ deals with the same files, we can obtain the target *e*-Service $e\mathcal{S}_0$ presented in Example 1 by composing the $e\mathcal{S}_1$ and $e\mathcal{S}_2$. This can be done by representing $e\mathcal{S}_0$, $e\mathcal{S}_1$, and $e\mathcal{S}_2$ as Situation Calculus theories Γ_0, Γ_1, Γ_2, construct from them Γ_C, and check its satisfiability. ■

Computing *e*-Service Composition

Next we turn to the problem of actually computing *e*-Service composition. To do so, we resort to a correspondence between the propositional Situation Calculus and Deterministic Propositional Dynamic Logic (DPDL) (Kozen & Tiuryn 1990). DPDL formulas are built by starting from atomic propositions and *deterministic* atomic actions as follows:

$$\phi \quad \longrightarrow \quad P \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \langle r\rangle\phi \mid [r]\phi$$

where P is an atomic proposition and r is a regular expression over the set of actions. 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. Let u be an abbreviation for $(\cup_{a\in\Sigma} a)^*$, then $[u]$ represents the *master modality*, which can be used to state universal assertions.

DPDL enjoys two properties that are of particular interest for our aims. The first is the *tree model property*, which says that every model of a formula can be unwound to a (possibly infinite) tree. The second is the *small model property*, which says that every satisfiable formula admits a finite model of size at most exponential in the size of the formula itself.

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

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

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 $[u](\bigwedge_{a\in\Sigma}\langle a\rangle\mathtt{true})$, and implicitly take into account the tree model property;
- to model the initial situation Φ_0, we add the conjunct $\delta(\Phi_0)$; [4]
- 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 $F(do(\bar{a},s)) \equiv \Phi_F^{\bar{a}}(s)$ – where $\Phi_F^{\bar{a}}(s)$ 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 $[u]([\bar{a}]F \equiv \delta(\Phi_F^{\bar{a}}(s)))$;
- for the last two axioms of Γ_C, we add the conjuncts $[u](Poss\_a \wedge Final \supset \bigvee_{i=1}^n Step\_a_i \wedge Poss\_a_i)$ and $[u](Final \supset \bigwedge_{i=1}^n Final_i)$.

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 2 *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. $\Box$

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 2 we get the following complexity result.

Theorem 3 *Checking the existence of an* e*-Service composition can be done in EXPTIME.*

[4] Note that $[u]\cdot$ does not appear in front of the propositional formula Φ_0.

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.

From a practical point of view, because of the correspondence between PDLs and Description Logics (Calvanese *et al.* 2001), one can use current highly optimized Description Logic systems (Baader *et al.* 2002)[5] to check the existence of *e*-Service compositions. Since these systems are based on tableaux techniques that construct a model when checking for satisfiability, one can, with minor modifications, also return the composition labeling.

Discussion

In this paper we have studied *e*-Services in an abstract framework, that of the execution trees, which has allowed us to avoid the peculiarities of any particular representational formalism. One of the main contributions of this paper is to clarify, at least under certain assumptions, the notion of composition in this very general setting. We have also argued that such an abstract view can indeed be realized in reasoning about actions formalisms, focusing in particular on the well-known Situation Calculus.

Note that our definition of *e*-Service composition does not require execution trees to be finite branching (i.e. to have only a finite set of possible interactions), but also allows for infinite branching execution trees. This opens up the possibility of having parametrized actions, whose parameters are arbitrary terms. To capture *e*-Services and *e*-Service composition in this case, one has to resort to the full (non-propositional) Situation Calculus. Observe that the logical theories that represent execution trees need to be complete (we have complete information on such trees) and that particular care must be taken in order to have terms denoting each action (i.e., one needs some form of infinite domain closure, expressible only in second-order logic). Naturally, in general, decidability of composition is lost in this case, and it becomes of interest to understand for which theories decidability is preserved.

In this paper we have made use of Situation Calculus, especially because it is one of the best known formalism for reasoning about actions. However, the basic ideas of this paper may be easily exported to other reasoning about actions formalisms. Of particular interest is looking at *e*-Service compositions, as defined here, in the framework proposed by (McIlraith, Son, & Zeng 2001; McIlraith & Son 2002). That is each *e*-Service, including the target *e*-Service, is represented by a GOLOG/CONGOLOG program (which indeed defines an execution tree, although possibly nondeterministic). Observe that one of the main differences between our approach an that in (McIlraith, Son, & Zeng 2001; McIlraith & Son 2002) is that in our approach the client's needs are themselves expressed by an *e*-Service, while in (McIlraith, Son, & Zeng 2001; McIlraith & Son 2002) they are expressed as customization conditions on available *e*-Services.

Finally, note that *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. It is interesting to notice that the notion of reuse has also arisen in planning. One of the best known contributions in this sense is that in (Firby 1987), where a planning approach is presented, based on fulfilling the goal by suitably selecting the right plan from a plan library. There, the plan was only selected from those in the plan library, and possibly customized to achieve the current goal. One could think of performing planning composition by reusing parts of the plans in the library, exactly following the ideas at the base of *e*-Service composition. The study in this paper can be a starting point for such a research.

Acknowledgments

This work has been partially supported by MIUR through the "Fondo Strategico 2000" Project *VISPO (Virtual-district Internet-based Service PlatfOrm)* (`http://cube-si.elet.polimi.it/vispo/index.htm`) and the "FIRB 2001" Project *MAIS (Multi-channel Adaptive Information Systems)*.

The work of Massimo Mecella has been also partially supported by the European Commission under Contract No. IST-2001-35217, Project *EU-PUBLI.com (Facilitating Co-operation amongst European Public Administration Employees)* (`http://www.eu-publi.com/`).

References

Aiello, M.; Papazoglou, M.; Yang, J.; Carman, M.; Pistore, M.; Serafini, L.; and Traverso, P. 2002. A request language for web-services based on planning and constraint satisfaction. In *Proc. of VLDB-TES'02*.

Baader, F.; Calvanese, D.; McGuinness, D.; Nardi, D.; and Patel-Schneider, P. F., eds. 2002. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press. To appear.

Calvanese, D.; De Giacomo, G.; Lenzerini, M.; and Nardi, D. 2001. Reasoning in expressive description logics. In *Handbook of Automated Reasoning*. Elsevier. 1581–1634.

Casati, F., and Shan, M. 2001. Dynamic and adaptive composition of *e*-services. *Information Systems* 6(3).

Fauvet, M.; Dumas, M.; Benatallah, B.; and Paik, H. 2001. Peer-to-peer traced execution of composite services. In *Proc. of VLDB-TES'01*.

Firby, J. 1987. An investigation into reactive planning in complex domains. In *Proc. of AAAI'87*.

Kozen, D., and Tiuryn, J. 1990. Logics of programs. In *Handbook of Theoretical Computer Science — Formal Models and Semantics*, 789–840. Elsevier.

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

McCarthy, J., and Hayes, P. J. 1969. Some philosophical problems from the standpoint of aritificial intelligence. *Machine Intelligence* 4:463–502.

McIlraith, S., and Son, T. 2002. Adapting golog for composition of semantic web services. In *Proc. of KR'02*.

McIlraith, S.; Son, T.; and Zeng, H. 2001. Semantic web services. *IEEE Intelligent Systems* 16(2).

Mecella, M.; Parisi Presicce, F.; and Pernici, B. 2002. Modeling *e*-service orchestration through petri nets. In *Proc. of VLDB-TES'02*.

Pilioura, T., and Tsalgatidou, A. 2001. *e*-services: Current technologies and open issues. In *Proc. of VLDB-TES'01*.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press.

Shegalov, G.; Gillmann, M.; and Weikum, G. 2001. XML-enabled workflow management for *e*-services across heterogeneous platforms. *VLDB Journal* 10(1).

Wodtke, D., and Weikum, G. 1997. A formal foundation for distributed workflow execution based on state charts. In *Proc. of ICDT'97*.