# $e$-Service Composition by Description Logics Based Reasoning

Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo,
Maurizio Lenzerini, and Massimo Mecella
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
*lastname*@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 propose a general framework addressing the problem of $e$-Service composition. We then show that, under certain assumptions, composition can be realized through DL-based reasoning.

## 1 Introduction

The spreading of network and business-to-business technologies [13] has changed the way business is performed, giving rise to the so called *virtual enterprises* and communities [8]. Companies are able to export services as semantically defined functionalities to a vast number of customers, and to cooperate by composing and integrating services over the Web. Such services, usually referred to as $e$-Services or Web Services, are available to users or other applications and allow them to gather data or to perform specific tasks.

Research on $e$-Services considers, as fundamental, service composition, i.e., how to compose and coordinate different services, to be assembled together in order to support more complex services and goals. Interestingly, many contributions on this issue come from the AI community [1, 11, 2, 15]. Despite the work done so far, composition is still largely unexplored and to the best of our knowledge an overall agreed upon comprehension of what $e$-Service and $e$-Service composition are, in an abstract and general fashion, is still lacking.

In this paper, we formalize the problem of composition in a general and formal framework. Then, we instantiate such a general framework to a specific formalism for Reasoning about Actions, that of Situation Calculus, and show that composition can be characterized as logical satisfiability under certain assumptions. Finally, resorting to DL basic reasoning, we provide algorithms for performing $e$-Services composition and show EXPTIME decidability of such a problem (under certain assumptions).

## 2 Framework

Generally speaking, an $e$-Service is a software artifact (delivered over the Internet) that interacts with its clients (possibly in a repeated way), which can be either human users or other $e$-Services [1, 11, 2, 15]. An interaction consists of a client invoking a command, i.e., an atomic action, and waiting for the fulfillment of the specific tasks and (possibly) the return of some information. Under certain circumstances, i.e., when the client has reached his goal, he may terminate the interactions. However, in principle, a given $e$-Service may need to interact with a client for an unbounded, or even infinite, number of steps, thus providing the client with a continuous service. Therefore, an $e$-Service can be characterized in terms of the sequences of actions it is able to execute, i.e., its behavior. In what follows, we refer to this conceptual vision of an $e$-Service as $e$-Service *schema*. 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.

When a client invokes an $e$-Service $E$, it may happen that $E$ does not execute all of its actions on its own, since it *delegates* some or all of them to other $e$-Services. All this is transparent to the client. To precisely capture such a situation, we introduce the notion of *community* of $e$-Services, which is formally characterized by: *(i)* a common set of actions, called the *alphabet* of the community; *(ii)* 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 of $e$-Services 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.

The behavior of an $e$-Service can be analyzed from two different points of view. From the external point of view, i.e., that of a client, an $e$-Service is seen as a "black box" that executes sequences of atomic *actions* with constraints on their invocation order. From the internal point of view, i.e., that of an application running an instance of $E$, it is important to specify whether each action is executed by $E$ itself or whether its execution is delegated to another $e$-Service belonging to the same community $C$ of $E$, transparently to the client of $E$. Therefore, it is natural to consider the $e$-Service schema as constituted by two different parts, called *external schema* and *internal schema*, abstractly[1] representing an $e$-Service from its external and its internal point of view, respectively.

The external schema specifies the behavior of an $e$-Service in terms of a tree of actions, called *external execution tree*. Each node $x$ of the tree represents the history of the sequence of interactions between the client and the $e$-Service executed so far. For every action $a$ that can be executed at the point represented by $x$, there is a (single) successor node $y_a$ with the edge $(x, y_a)$ labeled by $a$. $y_a$ represents the fact that, after performing the sequence of actions leading to $x$, the client chooses to execute the action $a$, among those possible. Some nodes of the execution tree are *final*: when a node is final, and only then, the client can end the interaction[2].

[1]We are not concerned with any specification formalism, here.

[2]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

The internal schema maintains, besides the behavior of the $e$-Service, the information on which $e$-Services in the community execute each given action of the external schema. Uniformly with the external schema, the internal schema is specified as an *internal execution tree.* Formally, each edge of an internal execution tree of an $e$-Service $E$ is labeled by $(a, I)$, where $a$ is the executed action and $I$ is a nonempty set of (identifiers of) $e$-Service instances[3]. The special instance identifier `this` indicates the actions that are executed by the running instance of $E$ itself.

An internal execution tree $t_i$ *conforms* to an external execution tree $t_e$ if $t_e$ is equal to the external execution tree obtained from $t_i$ by projecting out the part of the labeling denoting the $e$-Service instances.

The internal execution tree $t_i$ of an $e$-Service $E$ is *coherent* with a community $C$ if: *(i)* for each edge labeled with $(a, I)$, the action $a$ is in the alphabet of $C$, and for each $e'$ in $I$, $e'$ is an instance of a member of the community $C$; *(ii)* for each path $p$ in $t_i$ from the root of $t_i$ to a node $x$, and for each $e'$ appearing in $p$, where $e'$ is an instance, different from `this`, of an $e$-Service $E'$, the projection[4] of $p$ on $e'$ is a path in the external execution tree $t'_e$ of $E'$ from the root of $t'_e$ to a node $y$, and moreover, if $x$ is final in $t_i$, then $y$ is final in $t'_e$.

When a client 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 happen that *composite* $e$-Service, obtained by combining a *set* of available *component* $e$-Services, might be used. Composition deals with such a problem, namely *synthesizing* a new $e$-Service starting from available ones, thus producing a *composite* e-*Service specification.* In our framework, this formally correspond to say: given an $e$-Service community $C$ and the external execution tree $t_e$ of a target $e$-Service $E$ expressed in terms of the alphabet of $C$, synthesize an internal execution tree $t_i$ such that *(i)* $t_i$ conforms to $t_e$, *(ii)* $t_i$ delegates all actions to the $e$-Services of $C$, and *(iii)* $t_i$ is coherent with $C$. When such an internal tree exists we say that $E$ *can be composed* using $C$.

**Example 1** Figure 1(a) shows an external execution tree of an $e$-Service $E$ that allows for searching and buying `mp3` files. After an authentication step (action `auth`), in which the client provides *userID* and *password*, the $e$-Service asks for search parameters (e.g., author or group name, album or song title) and returns a list of matching files (action `search`); then, the client can: *(i)* select and listen to a song (action `listen`), and choose whether to perform another `search` or whether to add the selected file to the cart (action `add_to_cart`); *(ii)* `add_to_cart` a file without listening to it. Then, the client chooses whether to perform those actions again. Finally, by providing its payment method details the client buys and downloads the content of the cart (action `buy`).

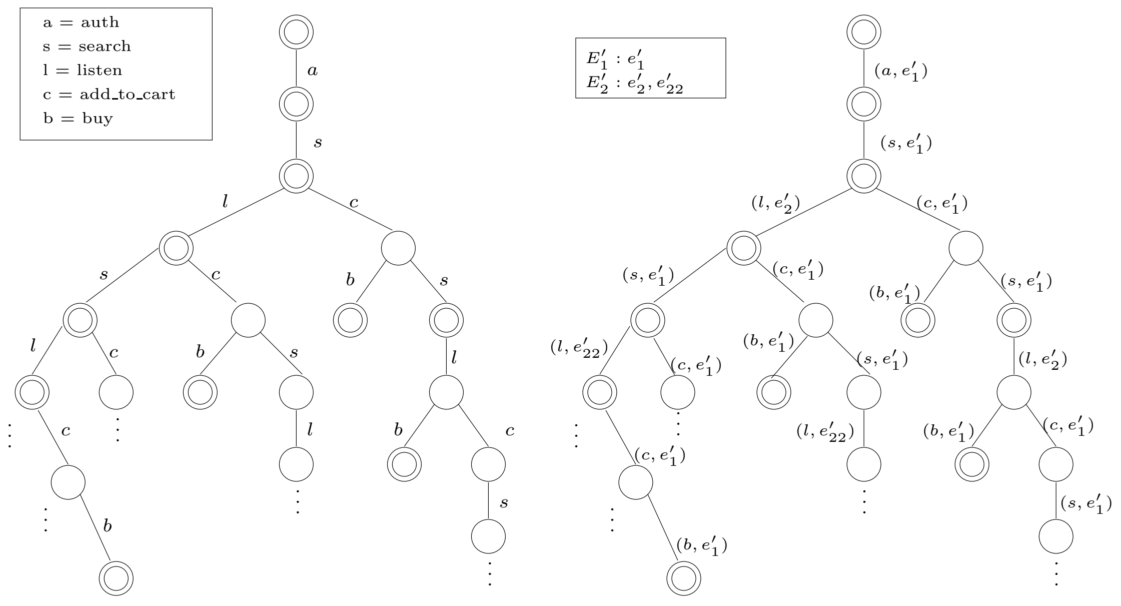
Figure 1(b)[5] shows an internal execution tree, conforming to the external execu-

mechanism as orthogonal to the $e$-Service specification.

[3]Note that the execution of actions labeling edges of the execution tree can be delegated in parallel to more than one $e$-Service instance.

[4]The *projection* of a path $p$ on an instance $e'$ of an $e$-Service $E'$ is 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 [4].

[5]Note that each action of $E$ is delegated to exactly one other instance. Hence, for simplicity, in the figure we have denoted a label $(a, \{e\})$ simply by $(a, e)$.



(a) external (b) internal

Figure 1: Example of $e$-Service schema

tion tree in Figure 1(a). In particular, the figure explicits the assignment of actions to component $e$-Services $E'_1$ and $E'_2$ of $e$-Service $E$, where, intuitively, $e$-Service $E'_1$ behaves like $E$ except that it allows only for (possibly) listening to a *sample* of a song: such an action does not appear in the execution tree of $E$; and $E'_2$ consists of the action `listen` done just once: therefore in $E$ this action is executed each time by a new instance. ■

## 3 Characterizing *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.[6] Therefore, in what follows, we address $e$-Services whose execution trees have a finite representation. Here, we propose to use formalisms developed for Reasoning about Actions to represent $e$-Services, and show that we can use logical reasoning, in particular, satisfiability, to characterize the problem of $e$-Service composition. This approach gives us the ability of dealing with a large class of $e$-Services, including those formalized by finite state machine-based formalisms such as UML statecharts, using a compact and high-level representation. There are many possible action languages that can be used for representing $e$-Services (including some tightly related to DL [7, 10]). Here we focus

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

on Reiter's Situation Calculus Basic Action Theories [14], 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 also make the following assumptions: *(i)* the action alphabet of the community is finite; *(ii)* for each $e$-Service there is only a fixed finite number of active instances, and, in fact, wlog, we assume that there is only one, so that we can omit the term "instance" when referring to an $e$-Service. Within this setting, in the next section, we show how to solve the composition problem. Instead, how to deal with an unbounded number of instances remains open for future work.

We will not go over the Situation Calculus 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 [14]:

- 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 of an $e$-Service $e\mathcal{S}$ as a Basic Action Theory $\Gamma$, where each action is represented by a Situation Calculus action. $\Gamma$ includes among its fluents a special fluent $Final$, denoting that the $e$-Service execution can stop in that situation. Also, $\Gamma$ 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 $\Gamma$ 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 [14] formed only by the actions that are possible (as specified by *Poss*) directly corresponds to the external 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 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 $\mathcal{T}^{e\mathcal{S}}$ generated by $\Gamma$ 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 $\mathcal{T}^{e\mathcal{S}}$ 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 $\Gamma_0$ be the theory for the target $e$-Service. 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 $e$-Service $e\mathcal{S}_i$ executes action $a$ in situation $s$. Formally, we construct a Situation Calculus theory $\Gamma_C$ formed by the union of the axioms below:

- $\Gamma_0$;
- $\Gamma_i'$, for $i = 1, \ldots, n$, where $\Gamma_i'$ is obtained from $\Gamma_i$:
  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 $\Gamma_C$ is not a Basic Action Theory. In $\Gamma_C$, 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 $\Gamma_C$. 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.

It can be shown that $\Gamma_C$ *(i)* characterizes all the internal execution trees that conform to the external execution tree generated by $\Gamma_0$; *(ii)* delegates all actions to $e\mathcal{S}_1, \ldots, e\mathcal{S}_n$; *(iii)* is coherent with $e\mathcal{S}_1, \ldots, e\mathcal{S}_n$. More precisely it can be shown that from each model of $\Gamma_C$ 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 $\Gamma_C$.

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

**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 $\Gamma_C$ be the theory defined as above. Then, $\Gamma_C$ is satisfiable if and only if $e\mathcal{S}_0$ can be composed using $e\mathcal{S}_1, \ldots, e\mathcal{S}_n$.*

## 4 Computing *e*-Service Composition

Next we turn to the problem of actually synthesizing a composite *e*-Service. To do so, we will re-express Situation Calculus Action Theories as an $\mathcal{ALU}$ knowledge base. $\mathcal{ALU}$ 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 $\mathcal{ALU}$ knowledge base is a set of inclusion assertions of the form

$$C_1 \sqsubseteq C_2$$

where $C_1, C_2$ are arbitrary $\mathcal{ALU}$ 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 $\mathcal{ALU}$, since concept satisfiability in a knowledge base is already EXPTIME-hard for $\mathcal{AL}$ and is EXPTIME-complete for $\mathcal{ALC}$ which includes $\mathcal{ALU}$ (see [3] for details).

$\mathcal{ALU}$ (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.

We define a mapping $\delta$ from (uniform) Situation Calculus formulas (wlog in negation normal form) with a free situation variable $s$ to boolean combination of concepts as follows:

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

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

Next, we define the $\mathcal{ALU}$ counterpart $\Delta_C$ of $\Gamma_C$ 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 $\Phi_0$, we add the assertion $\mathit{Init} \sqsubseteq \delta(\Phi_0)$, where *Init* is a new atomic concept denoting the initial situation;
- for each precondition axiom $\forall s.\mathit{Poss}(a,s) \equiv \Psi_a(s)$, we add the assertion $\delta(\mathit{Poss}(a,s)) \equiv \delta(\Psi_a(s))$; 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 $\Sigma$ 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 assertion $\forall \bar{a}.F \equiv \delta(\Phi_F^{\bar{a}}(s))$;
- for the last two axioms of $\Gamma_C$, we add the assertions $\mathit{Poss\_a} \wedge \neg \mathit{Final} \sqsubseteq \bigsqcup_{i=1}^{n} \mathit{Step\_a_i} \sqcap \mathit{Poss\_a_i}$ and $\mathit{Final} \sqsubseteq \bigsqcap_{i=1}^{n} \mathit{Final}_i$.

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

**Theorem 2** *The Init concept is satisfiable in the $\mathcal{ALU}$-counterpart $\Delta_C$ of $\Gamma_C$ if and only if $\Gamma_C$ is satisfiable.*

Observe that the size of $\Delta_C$ is at most equal to the size of $\Gamma_C$ times the number of actions in $\Sigma$. Hence, from the EXPTIME-completeness of concept satisfiability in $\mathcal{ALU}$ knowledge bases 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.*

Observe that, because of the small model property and the single successor property, if *Init* is indeed satisfiable in $\Delta_C$ 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.

From a practical point of view, one can use current highly optimized Description Logic systems [3, 9] 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 such a model, which correspond to the internal execution tree constituting the composition.

## 5 Conclusion

In this paper we have studied *e*-Services and their composition in an abstract framework, that of the execution trees, which on the one hand has allowed us to avoid the peculiarities of any particular representational formalism. Then we have instantiated our framework to a Propositional Situation Calculus setting, a well-known formalism for reasoning about actions. In such a setting we have given a characterization of the problem of finding a composition in terms of satisfiability of a certain action theory. Finally, resorting to a translation of such a Situation Calculus theory in a Description Logic we have shown that such a problem is EXPTIME, and that current tableaux based DL-reasoning procedures can be used to actually obtain the composition.

We want to observe that what our Propositional Situation Calculus setting can capture is essentially a description of *e*-Services given in terms of finite state machines (compactly represented by resorting on propositional fluents). This is a particularly interesting class of descriptions since it is one of the classes most commonly used to describe *e*-Services in the literature [12, 6, 5].

Developments of the work presented here can go in several directions. First, a main open question remains, namely whether composition in our setting is EXPTIME-hard. Second, among others, we mention two of them, both of which deal with incomplete information. First, we may relax the assumption that an *e*-Service that joins a community must declare exactly its executions in terms of the external execution tree, and instead accept that they give a partial description of such executions. This would correspond to having several –possibly infinite– external execution trees for an *e*-Service joining the community, and the community should use such an incomplete specification so as to be compatible with all possible external execution trees it represents. Second, it is also interesting to consider a setting where the target *e*-Service is underspecified, so that several external execution trees are compatible with it. In this case however one may assume that since the client of the community has not provided an exact specification of the external execution tree, then the community is free to choose any of the execution trees. Observe that these are very different way to deal with incomplete information, both of which of interest 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 *Proc. of the 3rd VLDB Int. Workshop on Technologies for* e-*Services (VLDB-TES 2002)*, 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 *Proc. of the 1st Int. Semantic Web Conf. (ISWC 2002)*, 2002.

[3] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2003.

[4] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. A foundational vision of $e$-services. In *Proc. of the CAiSE 2003 Workshop on Web Services, e-Business, and the Semantic Web (WES 2003)*, 2003.

[5] 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.

[6] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: A new approach to design and analysis of e-service composition. In *Proc. of the 12th Int. World Wide Web Conference (WWW 2003)*, 2003.

[7] G. De Giacomo and M. 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.

[8] D. Georgakopoulos, editor. *Proc. of the 9th Int. Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises (RIDE-VE'99)*, 1999.

[9] V. Haarslev and R. 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.

[10] C. Lutz and U. 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.

[11] S. McIlraith and T. Son. Adapting Golog for composition of semantic web services. In *Proc. of the 8th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2002)*, pages 482–493, 2002.

[12] 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", 2002. Available on line at `http://www.dis.uniroma1.it/~mecella/publications/mp_techreport_212002.pdf`.

[13] B. Medjahed, B. Benatallah, A. Bouguettaya, A. Ngu, and A. Elmagarmid. Business-to-business interactions: Issues and enabling technologies. *Very Large Database J.*, 12(1), 2003.

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

[15] J. Yang and M. Papazoglou. Web components: A substrate for web service reuse and composition. In *Proc. of the 14th Int. Conf. on Advanced Information Systems Engineering (CAiSE 2002)*, 2002.