A Foundational Framework for *e*-Services

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

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

Abstract. In this paper we propose a foundational vision of *e*-Services, in which we distinguish between the external behavior of an *e*-Service as seen by clients, and the internal behavior as seen by a deployed application running the *e*-Service. Such behaviors are formally expressed as execution trees describing the interactions of the *e*-Service with its client and with other *e*-Services. Using these notions we formally define *e*-Service composition in a general way, without relying on any specific representation formalism. We have also provide a classification of *e*-Services based on relevant properties of the execution trees.

1 Introduction

Since the last few years, we are witnessing a great change in business paradigms. Different companies are able to pool together their services, in order to offer more complex, value added products and services. Thanks to the spreading of network and business-to-business technologies [18], that makes services easily accessible to a vast number of customers, companies are able to cooperate in very flexible ways, giving rise to the so called *virtual enterprises* and communities [11, 9].

Inter-organization cooperation can be supported by Cooperative Information Systems (CIS's) [8]. Many approaches have been proposed for the design and development of CIS's: business process coordination and service-based systems [7], agent-based technologies and systems [6], schema and data integration techniques [19, 13]. In particular, the former approach focuses on cooperation among different organizations that export services as semantically defined functionalities; cooperation is achieved 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. *Service Oriented Computing* (SOC) is a new emerging model for distributed computing that enables to build agile networks of collaborating business applications distributed within and across organizational boundaries [1].

Cooperation of *e*-Services poses many interesting challenges regarding, in particular, composability, synchronization, coordination, correctness verification [25]. However, in order to address such issues in an effective and well-founded way, *e*-Services need to be formally represented.

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

Up to now, research on *e*-Services has mainly concentrated on three issues, namely *(i)* service description and modeling, *(ii)* service discovery and *(iii)* service composition, i.e., how to compose and coordinate different services, to be assembled together in order to support more complex services and goals.

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. In [20] an *e*-Service is described in terms of interface and implementation, through Activity State Machine Types (ASMT's), i.e., state machines which specify valid states of the *e*-Service and valid state transitions, caused either by operation requests or by internal transitions of the *e*-Service. In [12], *e*-Services are modelled as views of complex inter-organization processes, and in [17] *e*-Services are represented as statecharts.

As for discovery, in [22] *e*-Services are considered as the composition of sub-*e*-Services, thus modeled as a hierarchy of parts (expressing functionalities of *e*-Services), based on a common ontology. On the assumption that all descriptions of available *e*-Services are stored in a common directory, an algorithm that select the service that best fits a given description (i.e., the request for specific capabilities) is presented, based on similarity notions.

Composition addresses the situation when a client request cannot be satisfied by any available *e*-Service, whereas a *composite* *e*-Service, obtained by combining a *set* of available *component* *e*-Services, might be used. Composition involves two different issues: the one of *composing by synthesis* a new *e*-Service starting from available ones, thus producing a *composite* e-*Service specification*, and the one of enacting, i.e., instantiating and executing, the composite *e*-Service by correctly coordinating the component ones; the latter is often referred to as *orchestration*, and it is concerned with monitoring control and data flow among the involved *e*-Services, in order to guarantee the correct execution of the composite *e*-Service.

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 [24] the issue of service composition is addressed, in order to create composite services by re-using, specializing and extending existing ones; in [14] 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.

As far as orchestration is concerned, in [5] an *e*-Service that performs coordination of *e*-Services is considered as a (meta)*e*-Service that can be transparently invoked by clients. In [10] 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 service itself), in a decentralized way, through peer-to-peer interactions. In [21] coordination of *e*-Services is carried out by an enactment engine interpreting process schemas modeled as statecharts [23], and in [16] orchestration of *e*-Services is addressed by means of Petri Nets.

All the above mentioned works deal with different facets of service oriented computing, but unfortunately an overall agreed upon comprehension of what an e-Service is, in an abstract and general fashion, is still lacking. Nevertheless, *(i)* a framework for formally representing e-Services, clearly defining both specification (i.e., design-time) and execution (i.e., run-time) issues, and *(ii)* a definition of e-Service composition and its properties, are crucial aspects for correctly addressing research on service oriented computing.

In this paper, we concentrate on these issues, and propose an abstract framework for e-Services, so as to provide the basis for e-Service representation and for formally defining the meaning of composition. Specifically, Section 2 defines the framework, which is then detailed in Sections 3 and 4 by considering e-Service specification and run-time issues, respectively. Section 5 proposes some dimensions according to which classify composite e-Services, and Section 6 illustrates some examples highlighting the main characteristics of the proposed classification. Section 7 deals with composition, in particular by formally defining such a notion in the context of the proposed framework. Finally, Section 8 concludes the paper, by pointing out future research directions.

2 General Framework

Generally speaking, an e-Service is a software artifact (delivered over the Internet) that interacts with its clients, which can be either human users or other e-Services, by directly executing certain actions and possibly interacting with other e-Services to delegate to them the execution of other programs. In this paper we take an abstract view of such an application and provide a conceptual description of an e-Service 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* indicates *how* an e-Service is realized, in terms of software applications corresponding to the e-Service schema, deployed on specific platforms. Since this aspect regards the technology underlying the e-Service implementation, it goes beyond the scope of this paper and we do not consider it any more. We have mentioned it for completeness and because it forms the basis for the following one.
- 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 in isolation with respect to the others.

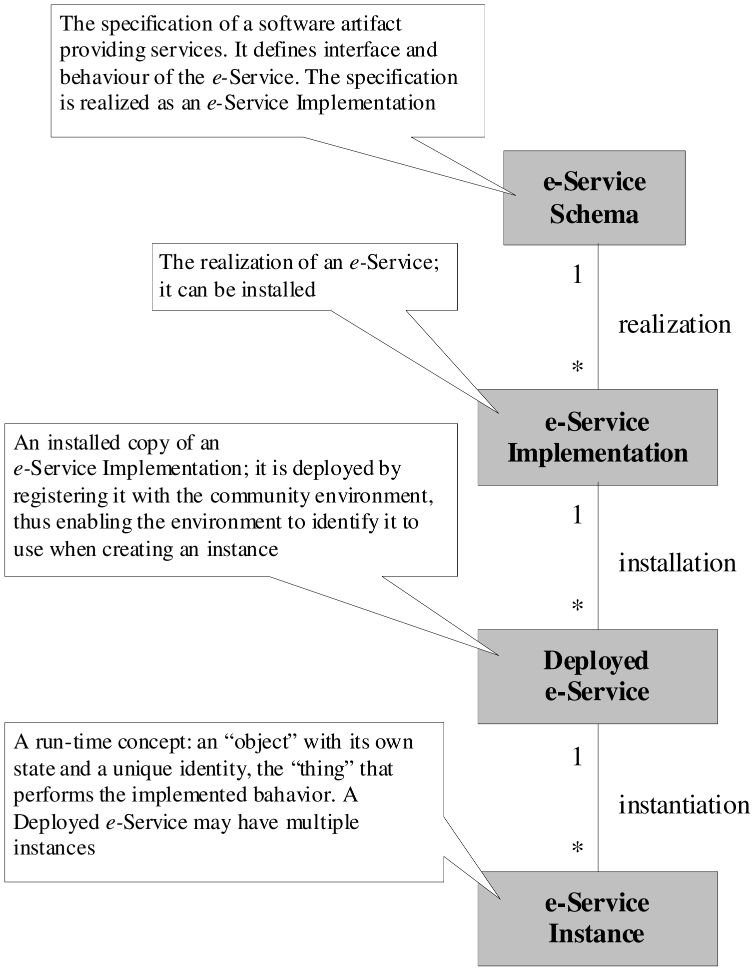

Fig. 1. Facets of an *e*-Service

As mentioned, the schema of an *e*-Service specifies what the *e*-Service does. From the external point of view of a client, the *e*-Service is seen as a black box that exhibits a certain "behavior", i.e., executes certain programs, which are represented as sequences of atomic *actions* with constraints on their invocation order. From the internal point of view, e.g., that of an application deploying an *e*-Service 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 with which E interacts, transparently to the client of E. To capture these two points of view we consider

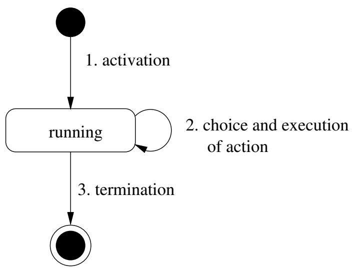

Fig. 2. Life cycle of an e-Service instance

the e-Service schema as constituted by two different parts, called *external schema* and *internal schema*, respectively representing an e-Service from the external point of view, i.e., its *behavior*, and from the internal point of view.

In order to execute an e-Service, the client needs to *activate* an instance from a deployed e-Service: the client can then interact with the e-Service instance by repeatedly *choosing* an action and waiting for the fulfillment of the specific task by the e-Service and (possibly) 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 and then is ready to execute new actions. Note that, in general, not all actions can be invoked at a given point: the possibility of invoking them depends on the previously executed ones, according to the external schema of the e-Service. Under certain circumstances, i.e., when the client has reached his goal, he may explicitly *end* (i.e., terminate) the e-Service instance. The state diagram in Figure 2 shows the life cycle of an e-Service instance.

Note that, 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. In this case, no operation for ending the e-Service is ever executed.

For an instance e of an e-Service E, the sequence of actions that have been executed at a given point and the point reached in the computation, as seen by a client, are specified in the so-called *external view* of e. Besides that, we need to consider also the so-called *internal view* of e, which describes also which actions are executed by e itself and which ones are delegated to which other e-Service instances, in accordance with the internal schema of E.

To precisely capture the possibility that an e-Service may delegate the execution of certain actions to other e-Services, we introduce the notion of *community* of e-Services, which is formally characterized by:

– a common set of actions, called 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 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 part of the service it provides to other members of the community. We call such an e-Service *composite*, whereas an e-Service that does not delegate the execution of any action is called *simple*. Also, the community may be used to generate (virtual) e-Services whose execution completely delegates actions to other members of the community.

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.

3 *e*-Service Schemas

As we already said, given an e-Service E belonging to a community C, the schema of E, describing the functional requirements of E, consists of two parts:

- the *external* schema of E, specifying the so called "behavior", i.e., the actions provided by E and the constraints on their invocation order;
- the *internal* schema of E, specifying which e-Services are going to execute each action of the behavior of E, taking into account that each action can be either executed by E itself or delegated to other e-Services of the community.

We now go into more details about the two schemas.

3.1 External Schema

The aim of the external schema is to abstractly express the behavior of the e-Service. To this end an adequate specification formalism must be used. In this paper 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*. 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. The node 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, thus getting to y_a. Therefore, each node represents a choice point at which the client makes a decision on the next action the e-Service should perform.

The root of the tree represents the fact that the client has not yet performed any interaction with the e-Service. Some nodes of the execution tree are *final*: when a node is final, and only then, the client can end the interaction. In other words, the execution of an e-Service can correctly terminate at these points[2].

Notably, an execution tree does not represent the information returned to the client, 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.

[2] 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.

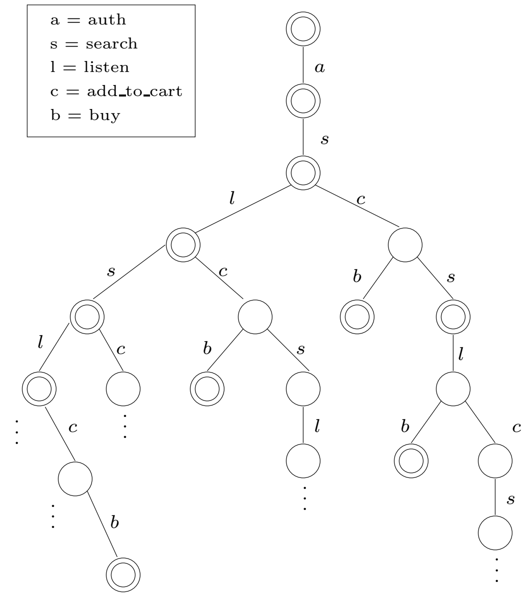

Fig. 3. Example of external execution tree of an *e*-Service

Example 1. Figure 3 shows an execution tree representing an *e*-Service that allows for searching and buying `mp3` files[3]. 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 (interaction `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 contents of the cart (action `buy`).

Note that, after the action `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 to some songs zero or more times. □

3.2 Internal Schema

The internal schema maintains, besides the behavior of the *e*-Services, the information on which *e*-Services in the community execute each given action of the external schema. As before, here we abstract from the specific formalism chosen for giving such a specification, instead we concentrate on the notion of internal

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

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 denoting the e-Services instances executing a. Every element of I is a pair (E', e'), where E' is an e-Service and e' is the identifier of an instance of E'. The identifier e' uniquely identifies the instance of E' within the internal execution tree. In 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 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_i we call *offered external execution tree* the external execution tree t_e obtained from t_i 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_i *conforms to* an external execution tree t_e if t_e is equal to the offered external execution tree of t_i. An e-Service is *well formed* if its internal execution tree conforms to 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 an internal execution tree t_i of an e-Service E, and a path p in t_i 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_i 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_i from the root of t_i 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'_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.

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, i.e., it is composite and hence delegates actions to other e-Service instances, the behavior of each one of such e-Service instances must be correct according to its external schema.

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

Example 2. Figure 4 shows an internal execution tree conforming to the external execution tree in Figure 3, where no action is delegated to other e-Service instances. Figure 5 shows a different internal execution tree, conforming again to the external execution tree in Figure 3, where the `listen` action is delegated to a different e-Service, using each time a new instance. In the examples each action is either executed by the running instance of E itself, or is delegated to

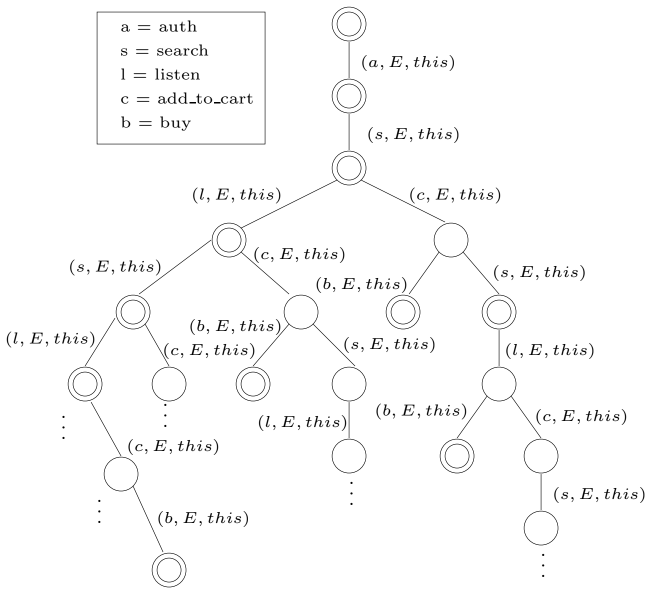

Fig. 4. Example of internal execution tree of a simple *e*-Service

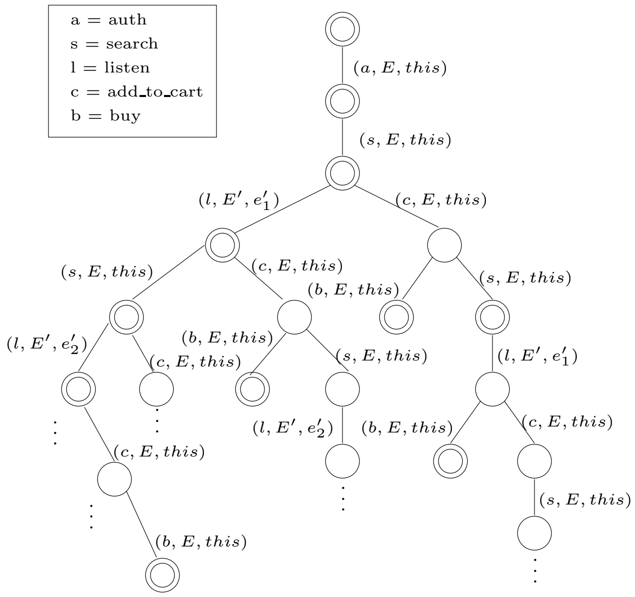

Fig. 5. Example of internal execution tree of a composite *e*-Service

exactly one other instance. Hence, for simplicity, in the figure we have denoted a label $(a, \{(E, e)\})$ simply by (a, E, e). $\square$

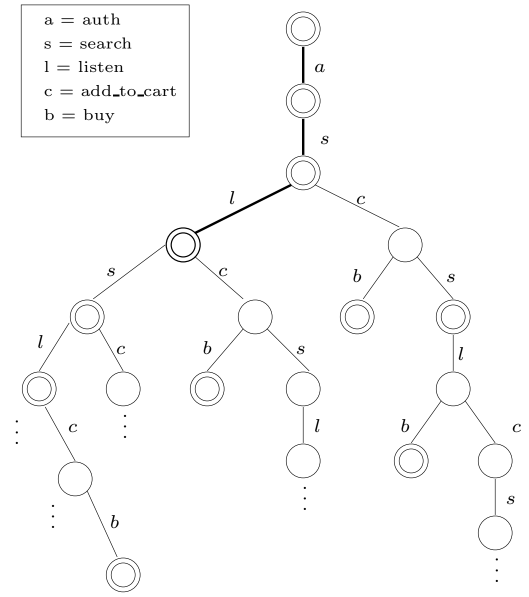

Fig. 6. External view of an *e*-Service instance

4 *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". The path from the root of the tree to the current position is the run of the *e*-Service so far, 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.

The external view characterizes the *e*-Service instance as seen by a client: the an execution has reached a certain point, the current position, and hence has a history of executed actions that led to that point and a choice of possible actions to do next, according to the external execution tree.

Example 3. Figure 6 shows an external view of an instance of the *e*-Service of Figure 3. 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 by a client that has provided its credentials and search parameters, has searched for and listened to one mp3 file, and has reached a point where it is necessary to choose whether *(i)* performing another search, *(ii)* adding the file to the cart, or *(iii)* terminating the e-Service (since the current position corresponds to a final node). The set of possible actions to do next are captured by the execution subtree having as root the current node. □

The internal view of an e-Service instance additionally maintains information on which e-Service instance executes which action. 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. In order for the e-Service instance to behave correctly, the various active instances are to be coordinated by an *orchestration engine* [4, 15]. The orchestration engine uses the internal view of the current instance and the external view of the other ones cooperating with it, so as to coordinate their execution, monitoring control and data flows, and guarantees that the composite e-Service behaves according to its (internal) execution tree. The orchestration engine is also in charge of instantiating and 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.

Note that the component e-Services can be, in their turn, composite. However, this aspect is transparent to the current orchestration engine since it acts as a client of the component e-Service instances, and hence uses only their external view.

5 Classification of e-Services

e-Services have many features, and they can be classified according to several dimensions. First of all, as we already observed, e-Services can be characterized as simple or composite: an e-Service is *composite* if it delegates some or all of its actions to other instances of e-Services in the community. 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-Services in order to completely fulfill the client's needs.

Composite e-Services can be classified according to their *delegation level*:

- a *partially delegating* composite e-Service executes some of the actions itself and possibly delegates other actions to component e-Services;
- a *fully delegating* composite e-Service delegates all offered actions to component e-Services.

Fully delegating composite e-Services are virtual, and often corresponds to organizations without a complex physical infrastructure; moreover, in some situations, organizations can also decide to only specify such e-Services and let them running on third party application service providers. Conversely, partially delegating composite e-Services should corresponds to organizations with some form of software infrastructure, as they realizes some operations themselves. By looking at the internal execution tree, the delegation level of an e-Service can be checked: if `this` is the only instance identifier occurring in the tree, then the e-Service is simple, if `this` occurs together with other instance identifiers, then the e-Service is a partially delegating composite e-Service, if `this` never occurs, then it is a fully delegating composite e-Service.

Composite e-Services can be further classified according to whether each action is executed by one or by more than one e-Service instance. 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 [3]. This property can be easily checked on the internal execution tree, by looking at whether for each label (a, I), the set I is a singleton or not.

Composite e-Services can be also classified according to whether the orchestration is interleaved or not:

- a composite e-Service E is *non-interleaving* if, whenever E activates an instance e' of a component e-Service E', it executes the component e-Service instance e' until such an instance terminates, without interleaving the actions of e' with the execution of other instances (either of E' of other component e-Services different from E'), including `this`;
- a composite e-Service is *interleaving* if it allows for interleaving the execution of actions by different component e-Service instances (including `this`). More in detail, an interleaving e-Service may delegate the execution of the same action to more than one component e-Service.

The opportunity of interleaving composite e-Services allow to develop more complex e-Services, in which information returned/sent to clients and/or different component e-Services can be aggregated and used to properly execute different commands. Whether an e-Service is interleaving or not can also be checked by looking at its internal execution tree: the e-Service is interleaving if and only if there is a path on which we find an instance e' followed by an instance e'', and on the path we find e' later on again. Note that, when a composite e-Service is non-interleaving, it executes the component e-Services as if they were atomic, i.e., it either executes them all or not at all.

Finally, composite e-Services can be characterized according to the *number of instances* of the component e-Services that can be created and that are simultaneously active. We distinguish the following cases:

1. for each component e-Service, at most one instance is used in the whole execution;
2. for each component e-Service E, more instances of E can be used, but before activating a new instance of E the previous one needs to be terminated;
3. for each component e-Service E, more instances of E can be used and can be simultaneously active.

Distinguishing composite e-Services on the basis of the used number of instances has some practical implications; in general, not all e-Services allow a client to activate more than one instance at the same time; moreover, especially in situations in which activating a new instance has a cost, composite e-Services should aim at using the minimum number of instances. Again, these properties can be verified on the internal execution tree: (1) holds if for each path there is at most one instance of each component e-Service; (2) holds if for each path the instances of a given e-Service are not interleaved; (3) does not pose any constraint on the instances of the component e-Services.

6 Running an e-Service Instance

In this section we first describe the basic, conceptual interaction protocol between an e-Service instance and its client, in terms of the correct sequence of interactions necessary to execute an e-Service instance. Then, we discuss some examples of interactions depending on the e-Service properties, covered in Section 5.

6.1 The Basic Protocol

In Section 2 we have briefly shown the steps that a client should perform in order to execute an e-Service:

1. activation of the e-Service instance
2. choice of the invokable actions
3. termination of the e-Service instance

where step (2) can be performed zero or more times, and steps (1) and (3) only once. Each of these steps is constituted by sub-steps, consisting in executing commands and in sending acknowledgements, each of them being executed by a different actor (either the client or the e-Service).

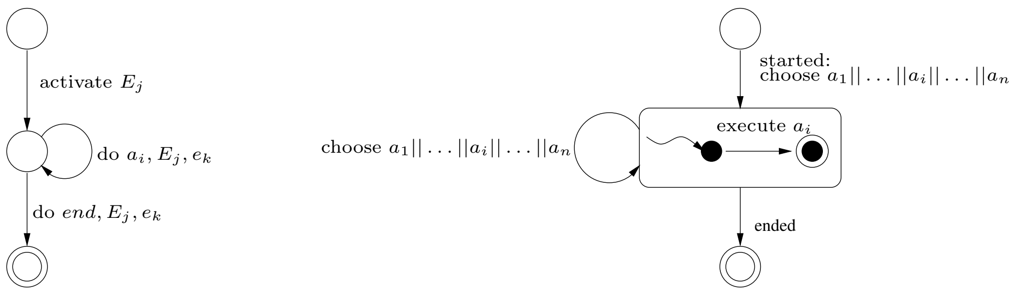

Fig. 7. Conceptual Interaction Protocol

For the sake of simplicity in what follows we describe the interactions between a client and an e-Service, assuming that no action is executed simultaneously by different e-Services (see Section 5). It is easy to extend what presented in order to cover also this case. Figure 7 shows the conceptual interaction protocol.

Activation. This step is needed to create the e-Service instance. The client[4] invokes the activation command, specifying the e-Service to interact with. If E_j is such an e-Service, the syntax of this command is:

$$\textbf{activate } E_j$$

When this command is invoked, all the necessary resources for the execution of a new instance e_k of e-Service E_j are allocated. Additionally, each e-Service instance creates a copy of both the internal and the external execution tree characterizing the e-Service schema it belongs to.

As soon as e_k is ready to execute, it responds to the client with the message

$$\textbf{started: choose } a_1||a_2\ldots||a_n$$

The purpose of this message is threefold. First, the client has an acknowledgement that the invoked e-Service has been activated and that the interactions may correctly start. Second, the client is informed about the instance identifier he will interact with (e.g., e_k). Third, the client is asked to choose the action to execute, among $a_1, \ldots, a_n$. The choice command is described next.

Choice. This step represents the interactions carried on between the client and the e-Service instance. Each e-Service instance is characterized, wrt the client, by its external execution tree, and all the actions are offered according to the information encoded in such a tree. Therefore, according to its external execution tree, the e-Service instance e_k proposes to its client a set of possible actions, e.g., $a_1, \ldots, a_n$, and asks the client to choose the action to execute next among $a_1, \ldots, a_n$. The syntax of this command is:

$$\textbf{choose } a_1||a_2||\ldots||a_i||\ldots||a_n$$

where $||$ is the choice symbol.

According to his/its goal, the client makes his/its choice by sending the message

$$\textbf{do } a_i, E_j, e_k$$

In this way, the client informs the instance e_k of e-Service E_j that he wants to execute next the action a_i. Once e_k has received this message, it executes action a_i. The execution of a_i is transparent to the client: the latter does not know anything about it, it only knows when it is ended, i.e., when the e-Service asks him/it to make another choice. This is shown in Figure 7 by the composite state that contains a state diagram modeling the execution of a_i.

The role of E_j and e_k becomes especially clear if we consider that the client could be a composite e-Service. When a composite e-Service E delegates an

[4] The client may be either a human user or another e-Service.

action to a component e-Service (e.g., E_j), it needs to activate a new e-Service instance (e_k), thus becoming in its turn a client. Therefore, on one side, E interacts with the *external* instances of the component e-Services, since E is a client of the latter; on the the other side, E chooses which action is to be invoked on which e-Service (either itself or a component e-Service) according to its internal execution tree, when E acts as "server" towards its client.

Termination. Among the set of invokable actions there is a particular action, **end**, which, if chosen, allows for terminating the interactions. Therefore, if the current node on the external execution tree is a final node, the e-Service proposes a choice as:

$$\textbf{choose}\ end||a_1||a_2||\ldots||a_i||\ldots||a_n$$

and if the client has reached his/its goal, he sends the message:

$$\textbf{do}\ end, E_j, e_k$$

The purpose of this action it to de-allocate all the resources associated with instance e_k of e-Service E_j. As soon as this is done, the e-Service informs its client of it with the message:

ended

6.2 Example of Interactions with Composite e-Services

In this section we discuss some examples of interactions between a composite e-Service and its client, being the composite e-Service a client for its component e-Services. Other examples are described in Appendix A. Each interaction is characterized in terms of the properties of the composite e-Service and of its component e-Services discussed in Section 5, i.e., the delegation level supported when interacting with a client, the way in which component e-Services can be composite (composition of component e-Services), and the number of active instances of each component e-Service. Not all of these properties have been addressed in the various research efforts. For example, to the best of our knowledge, the number of active instances has never been taken into account within composition. The purpose of this section is to describe such properties by means of examples, where they are combined together, in order to let the reader understand their importance within e-Service composition. For the sake of simplicity, in the examples, we make the following assumptions:

- The client shown in the leftmost column interacts with only one instance of the composite e-Service.
- The composite e-Service (shown in the central column) interacts with at most two instances of the component e-Service (s) (shown in the rightmost column). In other words, at most two instances of the same (or different) component e-Service are currently active.

Client C	e-Service E: instance e	e-Service E_1: instance e_1
1. **activate** E	2. **started:**	5. **started:**
3. **do** a_1, E, e	**choose** $end\|\|a_1\|\|a_2$	**choose** $end\|\|a_1\|\|a_3$
9. **do** a_5, E, e	4. **activate** E_1	7. **choose** $a_4\|\|a_5$
13. **do** a_7, E, e	6. **do** a_1, E_1, e_1	11. **choose** $end\|\|a_6$
21. **do** end, E, e	8. **choose** $a_4\|\|a_5$	15. **ended**
	10. **do** a_5, E_1, e_1	
	12. **choose** $end\|\|a_6\|\|a_7$	
	14. **do** end, E_1, e_1	e-Service E_2: instance e_2
	16. **activate** E_2	17. **started:**
	18. **do** a_7, E_2, e_2	**choose** $end\|\|a_7$
	20. **choose** $end\|\|a_8$	19. **choose** $end\|\|a_8$
	22. **do** end, E_2, e_2	23. **ended**
	24. **ended**	

Fig. 8. delegation level: fully delegation; composition of component e-Services: non-interleaved; number of active instances for each component e-Service: at most one in the whole execution.

Example 4. Figure 8 shows an example of interactions between an instance e of an e-Service E and its client C, where E is a composite e-Service obtained by executing two e-Services instances, e_1 and e_2, in a non-interleaving way, i.e., when e_1 (or e_2) is active, no action offered by another active e-Service instance is executed. E fully delegates the execution of actions to its component e-Services, therefore it does not offer any action. In other words, E can be seen as a "pure" orchestrator of e-Services. This notion of composite e-Service is not new: it can be found in [4], where the authors describe an engine for enacting an e-Service obtained by coordination of different e-Services.

Note that a composite e-Service E is a kind of "wrapper" of the component e-Service, indeed since its client interacts with the external execution tree of a composite e-Service, he is not aware whether he is interacting with a composite or a simple e-Service. Therefore, the interactions with the client C, involving the choice of which action to invoke next, have always the form "do a_i, E, e" where a_i represents any action chosen on instance e of e-Service E. Also, e forwards to C the actions offered by e_1 and e_2, and to e_1 and e_2 the requests of C.

Consider interaction 12, where component e-Service instance e_1 is active: since a component e-Service has to be executed in a non-interleaving way from its activation to its end, e can offer to its client the action a_7, offered by e-Service instance e_2, only if e_1 is (and can be) ended. Note that since the root of the execution tree is final, the end action belongs to the first set of offered actions. □

Example 5. Figure 9 shows an example of interaction when interleaving of component e-Services is allowed. Also, E partially delegates its actions to E_1. Given that, after activating E_1, at any time E can offer both its own actions and those offered by E_1.

Client C	e-Service E: instance e	e-Service E_1: instance e_1
1. **activate** E	2. **started:** **choose** $end\|\|a_1\|\|a_2$	7. **started:** **choose** $end\|\|a_1\|\|a_5$
3. **do** a_1, E, e	4. **choose** $a_4\|\|a_5$	9. **choose** $a_6\|\|a_7$
5. **do** a_5, E, e	6. **activate** E_1	17. **choose** $end\|\|a_{12}$
11. **do** a_8, E, e	8. **do** a_5, E_1, e_1	23. **ended**
15. **do** a_7, E, e	10. **choose** $a_6\|\|a_7\|\|a_8$	
19. **do** a_{13}, E, e	12. **choose** $a_6\|\|a_7\|\|a_{11}$	
21. **do** end, E, e	16. **do** a_7, E_1, e_1	
	18. **choose** $a_{12}\|\|a_{13}$	
	20. **choose** $end\|\|a_{12}$	
	22. **do** end, E_1, e_1	
	24. **ended**	

Fig. 9. delegation level: partial delegation; composition of component e-Services: interleaving; number of active instances for each component e-Service: at most one in the whole execution.

Consider interactions 9-12: in interaction 10 instance e offers actions a_6 and a_7 from e_1 and action a_8 from itself; since the client invokes a_7, the current position on the execution tree of e_1 does not change, therefore in the next interaction (12) e offers the same actions from e_1, i.e., a_6 and a_7. Note that this is not the case for the execution tree of the composite e-Service instance e, since it keeps track of *all* the operations that can be invoked through it, i.e., both those offered by e, and those offered by e_1, and therefore the current position on the execution tree of the composite e-Service always changes, also if the actions offered by e_1 are invoked.

Consider interaction 18: e_1 offers actions *end* and a_{12}, but e offers only a_{12} from e_1 (and a_{13} from itself). This is because the current position on the execution tree of e does not coincide with a final node. e offers the end action in interaction 20, when the current position on its execution tree corresponds to a final node. Finally, note that e correctly makes e_1 end, before ending itself. $\square$

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. Hence, a basic problem that needs to be addressed is that of e-Service *composition synthesis*, which can be formally described as follows: 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

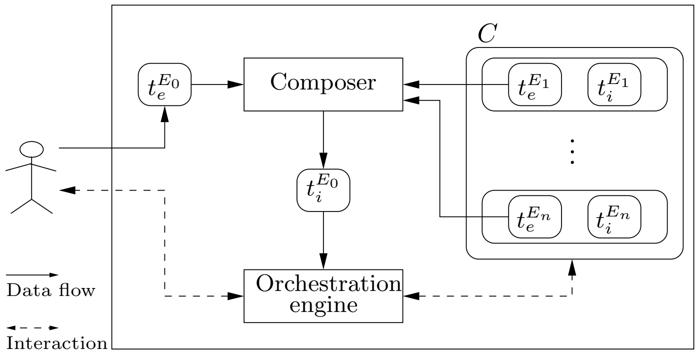

Fig. 10. *e*-Service Integration System

conforms to t_e, *(ii)* t_i delegates all actions to the e-Services of C (i.e., `this` does not appear in t_i, and *(iii)* t_i is coherent with C.

Obviously, depending on the characteristics of the composite and component e-Services (cfr. Section 5) many different forms of composition synthesis may be identified, and for each case appropriate techniques need to be devised. For example, [3] presents a technique for e-Service composition synthesis for the case of fully-delegating, interleaving composite e-Services in which for each component e-Service at most one instance is used, but each action may be delegated to more than one e-Service.

We point out that, to address what is typically referred to as e-Service *composition*, in fact one first needs to perform a composition synthesis, thus obtaining a composite e-Service fulfilling the client's needs, and then an instance of such a composite e-Service needs to be executed by an orchestration engine, as discussed in Section 4.

Figure 10 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_0, he presents his request in form of an external e-Service schema $t_e^{E_0}$ for E_0, and expects the e-Service Integration System to execute an instance of E_0. To do so, first the *composer* module makes the composite e-Service E_0 available for execution, by synthesizing an internal schema $t_i^{E_0}$ of E_0 that conforms to the external schema $t_e^{E_0}$ and is coherent with the community C. Then, using the internal schema $t_i^{E_0}$ as a specification, the *orchestration engine* activates an (internal) instance of E_0, and orchestrates the different available e-Services, by activating and interacting with them, so as to fulfill the client's needs. 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 Conclusions

In this paper we have proposed a conceptual, and formal, vision of *e*-Services, in which we distinguish between the external behavior of an *e*-Service as seen by clients, and the internal behavior as seen by a deployed application running the *e*-Service, which includes information on delegation of actions to other *e*-Services. Such a vision clarifies the notion of composition from a formal point of view. We have also provided a classification of *e*-Services based on relevant properties of the internal behavior.

Note that in the proposed framework, we have made the fundamental assumption that one has complete knowledge on the *e*-Services belonging to a community, in the form of their external and internal schema. We also assumed that a client gives a very precise specification (i.e., the external schema) of an *e*-Service he wants to have realized by a community. In particular, such a specification does not contain forms of "don't care" nondeterminism. Both such assumptions can be relaxed, and this leads to a development of the proposed framework that is left for further research.

Among other open issues, an important question concerns which *e*-Service maintains the *responsibility* with the end-client of executing an action. Throughout the paper we have implicitly assumed that the composite *e*-Service has control over the execution and maintains the responsibility, i.e., it wraps the component *e*-Services and the client interacts only with the invoked *e*-Service. Other scenarios are possible.

For example, the composite *e*-Service delegates both the control over the execution and the responsibility to a component *e*-Service. In other words, it simply act as broker among the end-client and the component *e*-Service.

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

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*, Proceedings of the 3rd VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2002), Hong Kong, Hong Kong SAR, 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*, Proceedings of the 1st International Semantic Web Conference (ISWC 2002), Chia, Sardegna, Italy, 2002.
3. D. Berardi, D. Calvanese, G De Giacomo, and M. Mecella, *Composing e-Services by Reasoning about Actions*, Proc. of the ICAPS 2003 Workshop on Planning for Web Services, 2003, To appear.
4. F. Casati, M. Sayal, and M.C. Shan, *Developing* e-*Services for Composing* e-*Services*, Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01), Interlaken, Switzerland, 2001.
5. F. Casati and M.C. Shan, *Dynamic and Adaptive Composition of* e-*Services*, Information Systems **6** (2001), no. 3.
6. J. Castro, M. Kolp, and J. Mylopoulos, *Towards Requirements-driven Information Systems Engineering: the Tropos Project*, Information Systems **27** (2002), no. 6.
7. U. Dayal, M. Hsu, and R. Ladin, *Business Process Coordination: State of the Art, Trends and Open Issues*, Proceedings of the 27th Very Large Databases Conference (VLDB 2001), Roma, Italy, 2001.
8. G. De Michelis, E. Dubois, M. Jarke, F. Matthes, J. Mylopoulos, M.P. Papazoglou, K. Pohl, J. Schmidt, C. Woo, and E. Yu, *Cooperative Information Systems: A Manifesto*, Cooperative Information Systems: Trends & Directions (M.P. Papazoglou and G. Schlageter, eds.), Accademic-Press, 1997.
9. A.K. Elmagarmid and W.J. McIver Jr, *The Ongoing March Towards Digital Government (Special Issue)*, IEEE Computer **34** (2001), no. 2.
10. M.C. Fauvet, M. Dumas, B. Benatallah, and H.Y. Paik, *Peer-to-Peer Traced Execution of Composite Services*, Proceedings of the 2nd VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2001), Rome, Italy, 2001.
11. D. Georgakopoulos (ed.), *Proceedings of the 9th International Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises (RIDE-VE'99)*, Sydney, Australia, 1999.
12. E. Kafeza, D.K.W. Chiu, and I. Kafeza, *View-based Contracts in an* e-*Service Cross-Organizational Workflow Environment*, Proceedings of the 2nd VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2001), Rome, Italy, 2001.
13. M. Lenzerini, *Data Integration: A Theoretical Perspective*, Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2002), Madison, WI, USA, 2002.
14. S. McIlraith and T. Son, *Adapting Golog for Composition of Semantic Web Services*, Proceedings of the 8th International Conference on Knowledge Representation and Reasoning (KR 2002), Toulouse, France, 2002.
15. M. Mecella, F. Parisi Presicce, and B. Pernici, *Modeling* e-*Service Orchestration Through Petri Nets*, Proceedings of the 3rd VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2002), Hong Kong, Hong Kong SAR, China, 2002.
16. M. Mecella and B. Pernici, *Building Flexible and Cooperative Applications Based on* e-*Services*, Technical Report 21-2002, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Roma, Italy, 2002, (available on line at: `http://www.dis.uniroma1.it/~mecella/publications/mp_techreport_212002.pdf`).
17. M. Mecella, B. Pernici, and P. Craca, *Compatibility of* e-*Services in a Cooperative Multi-Platform Environment*, Proceedings of the 2nd VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2001), Rome, Italy, 2001.

18. B. Medjahed, B. Benatallah, A. Bouguettaya, A.H.H. Ngu, and A.K. Elmagarmid, *Business-to-Business Interactions: Issues and Enabling Technologies*, VLDB Journal **12** (2003), no. 1.
19. E. Rahm and P.A. Bernstein, *A Survey of Approaches to Automatic Schema Matching*, VLDB Journal **10** (2001), no. 4.
20. H. Schuster, D. Georgakopoulos, A. Cichocki, and D. Baker, *Modeling and Composing Service-based and Reference Process-based Multi-enterprise Processes*, Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE 2000), Stockholm, Sweden, 2000.
21. G. Shegalov, M. Gillmann, and G. Weikum, *XML-enabled Workflow Management for* e-*Services across Heterogeneous Platforms*, VLDB Journal **10** (2001), no. 1.
22. W.J. van den Heuvel, J. Yang, and M.P. Papazoglou, *Service Representation, Discovery and Composition for* e-*Marketplaces*, Proceedings of the 9th International Conference on Cooperative Information Systems (CoopIS 2001), Trento, Italy, 2001.
23. D. Wodtke and G. Weikum, *A Formal Foundation for Distributed Workflow Execution Based on State Charts*, Proceedings of the 6th International Conference on Database Theory (ICDT '97), Delphi, Greece, 1997.
24. J. Yang and M.P. Papazoglou, *Web Components: A Substrate for Web Service Reuse and Composition*, Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE'02), Toronto, Canada, 2002.
25. J. Yang, W.J. van den Heuvel, and M.P. Papazoglou, *Tackling the Challenges of Service Composition in* e-*Marketplaces*, Proceedings of the 12th International Workshop on Research Issues on Data Engineering: Engineering E-Commerce/E-Business Systems (RIDE-2EC 2002), San Jose, CA, USA, 2002.

A Appendix

Next, we show some other examples of interactions.

Example 6. Figure 11 shows an example of interactions involving a composite e-Service instance e that partially delegates its actions, i.e., it offers both its own actions and actions actually offered by another e-Service instance (e.g., e_1). The other features are as in example 4.

Consider interactions 13-16: e_1 offers a choice between end and a_8, whereas e offers a choice between actions a_8, a_9, end, and the client chooses to execute action a_9.

- e offers end because the current position on both its execution tree and on its component e-Services' execution tree coincides with a final node.
- Given the client's choice and given the non-interleaving of composition, e has to first terminate the interactions with e_1 and then it can execute the action a_9. Indeed, e cannot offer its own actions (e.g., a_9) while e_1 is executing, but can do it only after e_1 has offered an end action.

Note that the same interaction protocol might take place if the delegation level was of type "fully delegation" and there were two or more active instances of component e-Service E_1. □

Example 7. Figure 12 shows an instance e of a composite e-Service E interacting with 2 simultaneously active instances of component e-Service E_1, namely e_1 and e_2.

Consider interaction 13: when the client chooses to execute a_3, both e_1 and e_2 can execute it. However, the instance that will execute a_3 is already decided and this information is encoded in the execution tree of E. In this case, a_3 is executed by e_1 through the command "do a_3, E_1, e_1", indeed it is necessary to specify which instance has to execute a_3. □

Client C	e-Service E: instance e	e-Service E_1: instance e_1
1. **activate** E	2. **started:**	7. **started:**
3. **do** a_1, E, e	**choose** $end\Vert a_1\Vert a_2$	**choose** $end\Vert a_1\Vert a_5$
5. **do** a_5, E, e	4. **choose** $a_4\Vert a_5$	9. **choose** $a_6\Vert a_7$
11. **do** a_7, E, e	6. **activate** E_1	13. **choose** $end\Vert a_8$
15. **do** a_9, E, e	8. **do** a_5, E_1, e_1	17. **ended**
19. **do** end, E, e	10. **choose** $a_6\Vert a_7$	
	12. **do** a_7, E_1, e_1	
	14. **choose** $end\Vert a_8\Vert a_9$	
	16. **do** end, E_1, e_1	
	18. **choose** $end\Vert a_{10}$	
	20. **ended**	

Fig. 11. delegation level: partial delegation; composition of component e-Services: non-interleaving; number of active instances for each component e-Service: at most one in the whole execution.

Client C	e-Service E: instance e	e-Service E_1: instance e_1
1. **activate** E	2. **started:**	5. **started:**
3. **do** a_1, E, e	**choose** $end \Vert a_1 \Vert a_2$	**choose** $end \Vert a_1 \Vert a_3$
9. **do** a_1, E, e	4. **activate** E_1	7. **choose** $a_3 \Vert a_4$
15. **do** a_3, E, e	6. **do** a_1, E_1, e_1	17. **choose** $a_5 \Vert end$
19. **do** a_4, E, e	8. **choose** $a_3 \Vert a_4 \Vert a_1$	25. **ended**
23. **do** end, E, e	10. **activate** E_1	
	12. **do** a_1, E_1, e_2	
	14. **choose** $a_3 \Vert a_4$	e-Service E_1: instance e_2
	16. **do** a_3, E_1, e_1	11. **started:**
	18. **choose** $a_5 \Vert a_3 \Vert a_4$	**choose** $end \Vert a_1 \Vert a_3$
	20. **do** a_4, E_1, e_2	13. **choose** $a_3 \Vert a_4$
	22. **choose** $a_5 \Vert end$	21. **choose** end
	24. **do** end, E_1, e_1	27. **ended**
	26. **do** end, E_1, e_2	
	28. **ended**	

Fig. 12. delegation level: partial delegation; composition of component e-Services: interleaving; number of active instances for each component e-Service: more than one used and simultaneously active.

Client C	e-Service E: instance e	e-Service E_1: instance e_1
1. **activate** E	2. **2. started:**	**5. started:**
3. **do** a_1, E, e	**choose** $end \Vert a_1 \Vert a_2$	**choose** $end \Vert a_1 \Vert a_3$
9. **do** a_6, E, e	4. **4. activate** E_1	**7. choose** $a_3 \Vert a_4$
11. **do** a_3, E, e	6. **6. do** a_1, E_1, e_1	**13. choose end**
17. **do** a_1, E, e	8. **8. choose** $a_3 \Vert a_4 \Vert a_6$	**15. ended**
23. **do** a_9, E, e	10. **10. choose** $a_3 \Vert a_4 \Vert a_7$	
25. **do** a_4, E, e	12. **12. do** a_3, E_1, e_1	
29. **do** a_5, E, e	14. **14. do** end, E_1, e_1	e-Service E_1: instance e_2
33. **do** end, E, e	16. **16. choose** $a_1 \Vert a_2 \Vert end$	**19. started:**
	18. **18. activate** E_1	**choose** $end \Vert a_1 \Vert a_3$
	20. **20. do** a_1, E_1, e_2	**21. choose** $a_3 \Vert a_4$
	22. **22. choose** a_9	**27. choose** a_5
	24. **24. choose** $a_3 \Vert a_4 \Vert a_{10}$	**31. choose** $a_6 \Vert end$
	26. **26. do** a_4, E_1, e_2	**35. ended**
	28. **28. choose** a_5	
	30. **30. do** a_5, E_1, e_2	
	32. **32. choose** $a_{11} \Vert end$	
	34. **34. do** end, E_1, e_2	
	36. **36. ended**	

Fig. 13. delegation level: partial delegation; comopsition of component e-Services: interleaved; number of active instances for each component e-Service: more than one used, at most one active.

Example 8. Figure 13 shows an instance e of a composite e-Service E interacting with two instances e_1 and e_2 of the same component e-Service E_1, that are active one at a time. Such instances are interleaved with e, that offers its own actions. Note that despite the fact that the two instances belong to the same e-Service schema, from a certain point onwards, they do not offer the same set of actions: indeed, since the client has made different choices on the actions to execute on them, they evolve differently, i.e., the path from the root to the current position on their execution tree is different.

This example highlights several aspects of interactions. Consider interactions 26-28, where e_2 offers action a_5 and the composite e-Service limits to "reflect" a_5 to its client. The client is forced to choose a_5, but, at the same time, it is the client that performs the choice: the composite e-Service cannot choose instead of its client.

It may happen that, during a certain number of interactions, the composite e-Service may decide to offer only part (or none) of the actions by the component e-Services. This occurs in interactions 21-24.

Finally, consider the situation when there is only one component e-Service, whose instances are active one at a time, and the composite e-Service fully delegates its operations to the component e-Service instances: it is straightforward to see that the latter cannot be interleaved, but can only be executed in a non-interleaved way.

$\square$