A Foundational Vision of *e*-Services

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

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

Abstract. 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.

1 Introduction

The spreading of network and business-to-business technologies [11] has changed the way business is performed, giving rise to the so called *virtual enterprises* and communities [7]. 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. *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 [13]. However, in order to address such issues in an effective and well-founded way, *e*-Services need to be formally represented.

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.

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

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

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* [6, 10], 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. In what follows, we concentrate on composition synthesis: orchestration techniques go beyond the scope of this paper.

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 [12] the issue of service composition is addressed, in order to create composite services by re-using, specializing and extending existing ones; in [9] composition of *e*-Services is addressed by using GOLOG. 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.

All such 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, 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, in order 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 describes the basic, conceptual interaction protocol between a running *e*-Service and its client. Section 6 deals with composition, in particular by formally defining such a notion in the context of the proposed framework. Finally, Section 7 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.

– The *e*-Service *schema* specifies the features of an *e*-Service, in terms of functional and non-functional requirements. Functional requirements represent *what* an *e*-Service does. All other characteristics of *e*-Services, such as those related to quality, privacy, performance, etc. constitute the non-functional requirements. In what follows, we do not deal with non-functional requirements, and hence use the term "*e*-Service schema" to denote the specification of functional requirements only.

- The *e*-Service *implementation and deployment* indicate *how* an *e*-Service is realized, in terms of software applications corresponding to the *e*-Service schema, deployed on specific platforms. 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[2]. 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 independently from the others.

As mentioned, the schema of an *e*-Service specifies what the *e*-Service does. From the external point of view, i.e., that 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, i.e., 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 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. 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. 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

[2] Similarly, recovery mechanisms are outside the scope of this paper.

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(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. We call such an e-Service *composite*. If this is not the case, an e-Service is called *simple*. Simple e-Services realize offered actions directly in the software artifacts implementing them, whereas composite e-Services, when receiving requests from clients, can invoke other e-Services in order to completely fulfill the client's needs.

The community may also 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

In what follows, we go into more details about the two schemas introduced in the previous section.

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, which allows for a finite representation of such a behavior[3]. 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

[3] Typically, finite state machines are used [8, 5].

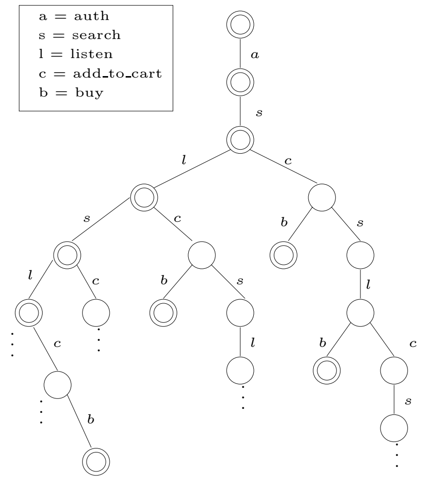


Fig. 1. Example of external execution tree of an *e*-Service

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[4].

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.

Example 1. Figure 1 shows an execution tree representing an *e*-Service that allows for searching and buying `mp3` files[5]. 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 content 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

[4] 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.

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

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

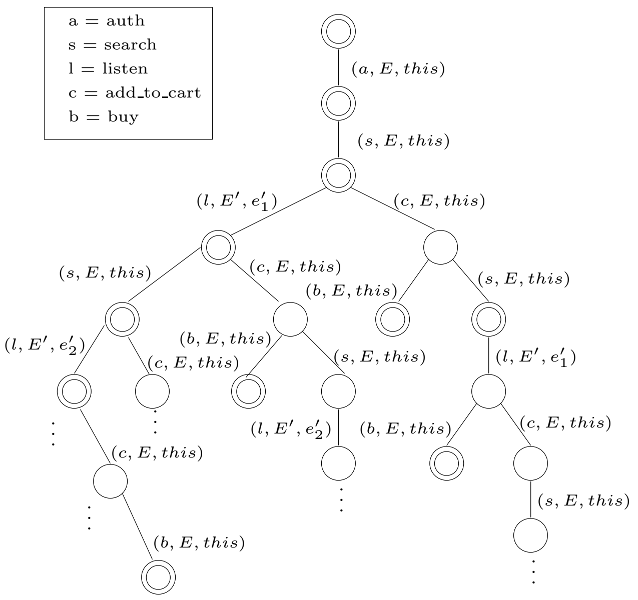


Fig. 2. Example of internal execution tree of a composite e-Service

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

Example 2. Figure 2 shows an internal execution tree, conforming to the external execution tree in Figure 1, where the `listen` action is delegated to a different e-Service, using each time a new instance. The internal execution tree, conforming again to the external execution tree in Figure 1, where no action is delegated to other e-Service instances, is characterized by the edges labeled by $(\alpha, E, this)$, being α any action.

In the examples each action is either executed by the running instance of E itself, or is delegated to exactly one other instance. Hence, for simplicity, in the figure we have denoted a label $(a, \{(E, e)\})$ simply by (a, E, e). $\square$

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", i.e., the point reached by the execution. The path from the root of the tree to the current position is the run of the e-Service so far, while the execution (sub-)tree

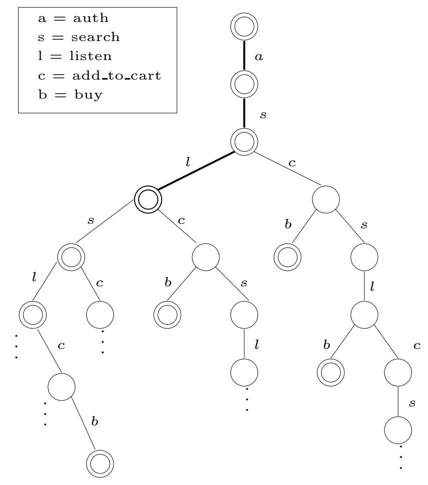


Fig. 3. External view of an *e*-Service instance

having as root the current position describes the behavior of what remains of the *e*-Service once the current position is reached.

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

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

Example 3. Figure 3 shows an external view of an instance of the *e*-Service of Figure 1. 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). □

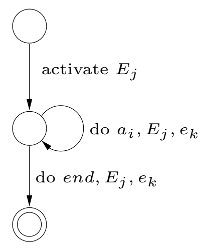
The internal view of an *e*-Service instance additionally maintains information on which *e*-Service instances execute which actions. At each point of the execution there may be several other active instances of *e*-Services that cooperate with the current one, each identified by its instance identifier. Note that, in general, an action can be 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 [4].

5 Running an *e*-Service Instance

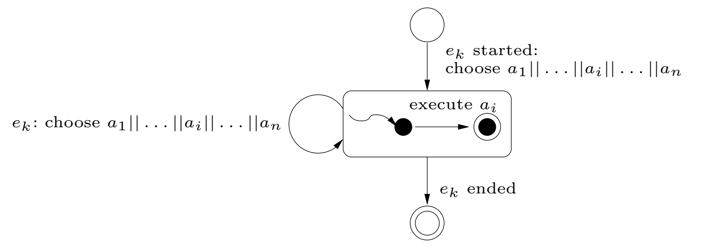
In Section 2 we have briefly shown the steps that a client should perform in order to execute an *e*-Service, namely:

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).



(a) Client


e_k started:
choose $a_1||\dots||a_i||\dots||a_n$
e_k: choose $a_1||\dots||a_i||\dots||a_n$
execute a_i
e_k ended

(b) *e*-Service

Fig. 4. Conceptual Interaction Protocol

In what follows we describe the correct sequence of interactions between a client and an *e*-Service, assuming, for the sake of simplicity, that no action is executed simultaneously by different *e*-Services (see Section 4). It is easy to extend what presented in order to cover also this case. Figure 4 shows the conceptual interaction protocol.

Activation. This step is needed to create the e-Service instance. The client[6] 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

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

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_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:

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

where $||$ is the choice symbol.

According to his goal, the client makes his 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 to make another choice. This is shown in Figure 4 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 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-Service, 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.

[6] The client may be either a human user or another e-Service, however, for the sake of simplicity, in what follows we consider a human 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:

$$e_k\textbf{: choose } end||a_1||a_2||\dots||a_i||\dots||a_n$$

and if the client has reached his 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:

$$e_k\textbf{: ended}$$

Examples of interactions can be found in [3].

6 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 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.

Figure 5 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 their external view, so as to fulfill the client's needs. The orchestration engine is also in charge of terminating the execution of component e-Service instances, offering the correct set of actions to the client, as defined by the external execution tree, and invoking the action chosen by the client on the e-Service that offers it.

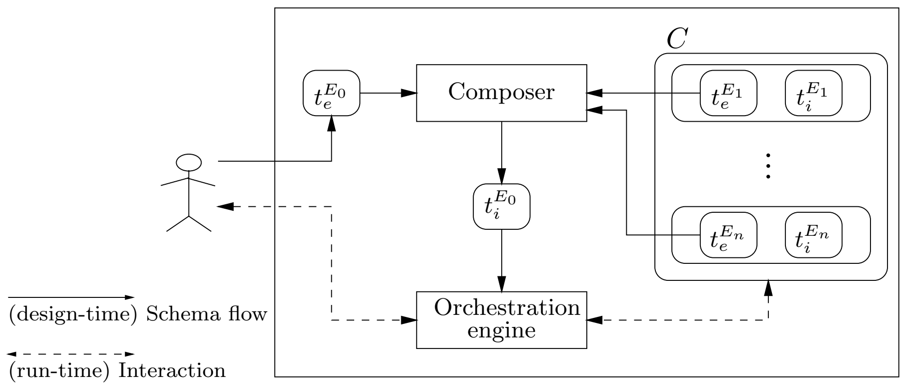


Fig. 5. *e*-Service Integration System

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.

7 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. On the basis of such a framework, we will study techniques for automatic composition synthesis.

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.

Acknowledgments

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

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, 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, M. Lenzerini, and M. Mecella, *A Foundamental Framework for e-Services*, Technical Report 10-03, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Roma, Italy, 2003, (available on line at: `http://www.dis.uniroma1.it/~berardi/publications/techRep/TR-10-2003.ps.gz`).
4. 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.
5. D. Berardi, L. De Rosa, F.and De Santis, and M. Mecella, *Finite State Automata as Conceptual Model for e-Services*, Proc. of the IDPT 2003 Conference, 2003, To appear.
6. F. Casati and M.C. Shan, *Dynamic and Adaptive Composition of* e-*Services*, Information Systems **6** (2001), no. 3.
7. 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.
8. R. Hull, M. Benedikt, V. Christophides, and J. Su, *E-Services: A Look Behind the Curtain*, Proceedings of the 22nd ACM SIGACT-SIGMOND-SIGART Symposium on Principles of Database Systems (PODS), June 2003.
9. 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.
10. 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`).
11. 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.
12. 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.
13. 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.