Synthesis of Composite *e*-Services based on Automated Reasoning

D. Berardi and **D. Calvanese** and **G. De Giacomo** and **M. Lenzerini** and **M. Mecella**
Dipartimento di Informatica e Sistemistica,
Università di Roma "La Sapienza"
`lastname@dis.uniroma1.it`

Introduction

e-Services represent a new model in the utilization of the network, in which self-contained, modular applications can be described, published, located and dynamically invoked, in a programming language independent way. This model, sometimes called *Service Oriented Computing* (SOC (Papazoglou & Georgakopoulos 2003)), enables building agile networks of collaborating business applications, distributed within and across organizational boundaries.

Research on *e*-Services spans over many interesting issues, including description, discovery, composition, synchronization, coordination, and verification (Hull *et al.* 2003). We are specifically interested in automatic *e*-Service composition. *e*-Service *composition* addresses the situation when a client request cannot be satisfied by any available *e*-Service, but a *composite e*-Service, obtained by combining "parts of" available *component e*-Services, might be used. Composition involves two different issues. The first, sometimes called *composition synthesis*, or simply *composition*, is concerned with synthesizing a new composite *e*-Service based on a set of available *e*-Services and the specification of a client request (called client specification). The synthesis process produces a specification of how to coordinate the component *e*-Services to obtain the composite *e*-Service that satisfies the client request. Such a specification can be produced either *automatically*, i.e., using a tool that implements a composition algorithm , or *manually* by a human. The second, often referred to as *orchestration*, is concerned with coordinating the various component *e*-Services according to some given specification, and also monitoring control and data flow among the involved *e*-Services, in order to guarantee the correct execution of the composite *e*-Service, synthesized in the previous phase.

Our research focuses on automatic composition synthesis. More specifically, we have devised techniques that, given (*i*) a client specification expressed as a transition system and (*ii*) a set of available *e*-Services, described as transition systems, synthesizes a composite *e*-Service that (*i*) uses only the available *e*-Services and (*ii*) interacts with the client "in accordance" with the input specification.

In fact, such a problem can be seen as an advanced form of Planning. In particular, as in Planning, the problem we are solving is the *synthesis of a program* of a specific form (not simple sequences of actions as in traditional planning, however, but more general transition systems); also, as in Planning, we have a *goal* that we want to realize (not a reachability goal, however, but the realization of a branching behavior as specified by the client); also has in Planning we have *constraints on the object that we synthesize* (not simple operators, however, but component transition systems that suitably orchestrated realize our goal).

This similarity to Planning also leads to related techniques in solving our problem. In particular our techniques are logic-based (as in deductive planning, but based on satisfiability instead of deduction). Specifically, our techniques are based on reducing the problem of checking the existence of a composition into concept satisfiability in a knowledge base expressed in a Description Logic (DL) (Baader *et al.* 2003) –or equivalently satisfiability of a formula in a theory expressed in a variant of Propositional Dynamic Logic (PDL) (Harel, Kozen, & Tiuryn 2000). With this reduction, reasoning tools for DLs can be directly used for composition synthesis, in particular by extracting a composite *e*-Service from a model of the DL knowledge base.

We have developed (and currently continue to extend) an open source prototype system[1], that realizes our techniques, which is, at the best of our knowledge, the first effective tool for automatic composition synthesis of *e*-Services that export their behavior.

e-Service Model

An *e*-Service is a software artifact that interacts with its client and possibly other *e*-Services in order to perform a specified task. A client can be either a human or a software application. When executed, an *e*-Service performs a given task by executing certain actions in coordination with the client or other *e*-Services. Specifically, each action in the task has a (single) *initiator*, typically the client, which requests the execution of the action possibly passing along information, and one or more *servants*, which are *e*-Services that respond to the request, possibly exchanging with the initiator further information. We build on the approach of (Berardi *et al.* 2003b; 2003a; 2003d;

[1]cfr. the PARIDE (Process-based frAmewoRk for composItion and orchestration of Dinamyc *E*-Services) Open Source Project: `http://sourceforge.net/projects/paride/` that is the general framework in which we intend to release the various prototypes produced by our research.

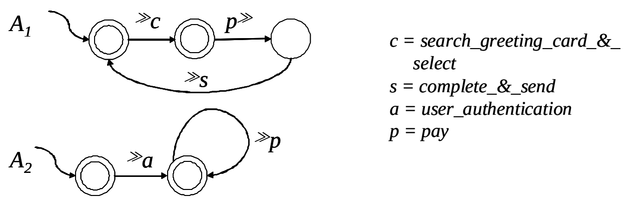

Figure 1: *e*-Services of the community

2003c), and characterize the exported behavior of an *e*-Service by the set of (possibly infinite) sequences of actions that the *e*-Service participates in, annotated with the role (either initiator or servant) the *e*-Service takes in executing each action. In order to represent such a role, we adorn each action symbol in the execution tree as follows: if the *e*-Service is one of the servants of an action a, then the action appears as $\gg a$, conversely if the *e*-Service is the initiator of a, then the action appears as $a^{\gg}$.

The set of (annotated) sequences of actions of the *e*-Service can be represented, by collapsing common prefixes, as a (possibly infinite) *execution tree*. Observe that in such an execution tree, for each node we can have at most one successor node for each annotated action. We annotate the nodes of the execution tree with the information on when a sequence of actions from the root to the node can be considered a completed execution of the *e*-Service, in the sense that the *e*-Service can terminate.

To represent the set of *e*-Services available to a client, we introduce the notion of *community* $\mathcal{C}$ of *e*-Services, which is a (finite) set of *e*-Services that share a common (finite) set of actions Σ, also called the *alphabet* of the community. Hence, to join a community, an *e*-Service needs to export its behavior in terms of the alphabet of the community.

We concentrate on *e*-Services whose behavior can be represented using a *finite number of states*. We do not consider any specific representation formalism for representing such states (such as action languages, situation calculus, statecharts, etc.). Instead, we use directly deterministic finite state machines (i.e., deterministic and finite labeled transition systems).

Given an *e*-Service A_i, the execution tree $T(A_i)$ *generated* by A_i is obtained by following in all possible ways the transitions of A_i, and annotating as final those nodes corresponding to the traversal of final states.

Example 1 In Figure 1 is shown a community of *e*-Services A_1, A_2. A user would like to send an e-card and, after a payment is requested (and obtained) by the *e*-Service A_1, the user can complete the e-card and send it. A_1 repeatedly is servant for searching an e-card and selecting one among those returned (`search_greeting_card_&_select`), is initiator for a `pay` action and servant for writing and sending the e-card (`complete_&_send`). A_2, after validating a (registered) user information (e.g., name, credit card number, account number, ...) (action `user_authentication` it is servant of), repeatedly is ready to act as servant of a `pay` action. ∎

e-Service Composition

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 be possible to suitably orchestrate (i.e., coordinate the execution of) the *e*-Services of the community so as to provide the service requested by the client. In other words, there may be an orchestration that coordinates both the initiator and the servants of each action, using the *e*-Services in the community, and that realizes what requested by the client.

Let the community $\mathcal{C}$ be formed by n *e*-Services $A_1, \ldots, A_n$, and let the service requested by the client be denoted by A_0. An *orchestration* O (also called composite *e*-Service) of the *e*-Services in $\mathcal{C}$ can be formalized as a so-called *orchestration tree* $T(O)$, that is an execution tree in which each *edge* of the tree is labeled by a triple (I, a, S), where a is the orchestrated action, $I \in [0..n]$ denotes the initiator, and $S \subseteq [1..n]$ denotes the nonempty set of *e*-Services in $\mathcal{C}$ that act as servants. As an example, the label $(0, a, \{1, 3\})$ means that the action a is initiated by the client and served by the *e*-Services A_1 and A_3.

Given an orchestration tree $T(O)$ and a path p in $T(O)$ starting from the root, we call the *projection* of p on an *e*-Service A_i the path obtained from p by:

- removing each edge whose label (I, a, S) is such that $i \notin \{I\} \cup S$, and collapsing start and end node of each removed edge;
- replacing, for each edge labeled by (I, a, S), with $I = i$, the label with $a^{\gg}$;
- replacing, for each edge labeled by (I, a, S), with $i \in S$, the label with $\gg a$.

We say that an orchestration O is *coherent* with a community $\mathcal{C}$ if for each path p in $T(O)$ from the root to a node x and for each *e*-Service A_i of $\mathcal{C}$, the projection of p on A_i is a path in the execution tree $T(A_i)$ from the root to some node y, and moreover, if x is final in $T(O)$, then y is final in $T(A_i)$.

We define as *client specification* a specification of the orchestration tree that the client would like to have realized using the *e*-Services in the community. Of the orchestration tree the client only specifies the actions, and whether it is the initiator of an action or not. Notably, we allow for incomplete information on the tree specified by the client. In other words the client may underspecify the sequences of actions of which the client is not the initiator, allowing the orchestrator to fill in the details left unspecified. Moreover the client may allow for don't care nondeterminism in the specification.

We consider underspecified specifications that can be expressed using a finite number of states, specifically as *non-deterministic* FSM in which the alphabet includes a special τ action that represents a finite sequence of actions in which the client is not the initiator (nor a servant). The nondeterministic FSM $\mathcal{A}_0$ specifies a set $\mathcal{T}(\mathcal{A}_0)$ of orchestration trees, and the client requires the orchestrator to realize one (any one) among such trees. Specifically, each orchestration tree in $\mathcal{T}(\mathcal{A}_0)$ is obtained by

- unfolding the FSM and while doing so, resolving the nondeterminism by choosing a single successor state for each

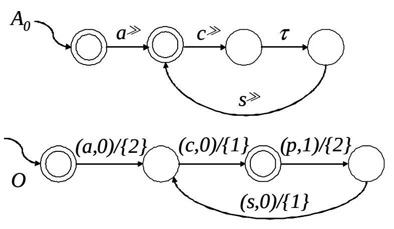

Figure 2: Client specification and orchestration

transition (including τ transitions); this generates a (deterministic), possibly infinite tree, whose edges are labeled by Σ_0 and whose nodes corresponding to final states of $\mathcal{A}_0$ are annotated as final;

- replacing each edge labeled by $a^{\gg}$ with an edge labeled by $(0, a, \cdot)$; this means that in the orchestration the client is the initiator of a;
- replacing each edge labeled by τ with a finite sequence of edges, each one labeled by $(j, a, \cdot)$, where a is some action, and $j \in [1..n]$; this means that for a τ action the orchestration is free to specify a finite sequence of interactions initiate by whatever *e*-Service except for the client;
- choosing for each edge a set of servants, and adding it to the label of the edge.

We say that an orchestration O *realizes* a client specification $\mathcal{A}_0$ if $O \in \mathcal{T}(\mathcal{A}_0)$. Given a community $\mathcal{C}$ of *e*-Services, and a client specification A_0, the problem of *composition existence* is the problem of checking whether there exists an orchestration that is coherent with $\mathcal{C}$ and that realizes A_0. The problem of *composition synthesis* is the problem of synthesizing an orchestration that is coherent with $\mathcal{C}$ and that realizes A_0. Since we are considering *e*-Services that have a finite number of states, we would like also to have an orchestration that can be represented with a finite number of states, i.e., as a Mealy FSM (MFSM).

Example 2 Figure 2 shows a possible client specification $\mathcal{A}_0$, which specifies that the client would like to act as initiator of a `user_authentication` action, followed by a `search_greeting_card_&_select` action. At this point the client allows the orchestration to act in a way not requiring interaction with the client itself, and then is interested to act as initiator of a `complete_&_send` action. O is the MSFM of an orchestration coherent with the *e*-Services of Example 1 and realizing the client specification $\mathcal{A}_0$. The orchestration specifies the client as initiator of the action `user_authentication` with A_1 as servant, then specifies the client as initiator of the action `search_greeting_card_&_select` with A_2 as servant; at this point, the orchestration specifies A_1 as initiator of the action `pay` served by A_2 (note that, correctly, the client is not involved, as specified by the τ action in $\mathcal{A}_0$), and finally the orchestration specifies the client as initiator of the action `complete_&_send` with A_1 as servant. ∎

Composition Synthesis

We address the problem of composition existence and synthesis in the FSM-based framework introduced above, by reducing the problem of composition existence to satisfiability of a concept in a knowledge base expressed in the well known Description Logic (DL) $\mathcal{ALCQ}_{reg}$ (Calvanese & De Giacomo 2003).

By exploiting reasoning methods for DLs based on model construction, such as tableaux algorithms, one can actually construct the MFSM orchestration. Notice that such algorithms need to be able to deal with reflexive transitive closure, introduced in $\mathcal{K}$ due to τ transitions in the client specification. If such τ transitions are not present (while the client specification may still be underspecified), one can resort to state-of-the-art implemented DL systems, such as FaCT and Racer, to check existence of a composition, while these systems are not yet usable for the actual construction of the MFSM orchestration, since they do not return the constructed model. A mentioned we developed an open source prototype, implemented in Java, that actually generates a composition (not supporting transitive closure yet) by building a DL model and extracting from it the MSFM that constitute the composition.

Acknowledgement This work has been supported by MIUR through the “FIRB 2001” Project *MAIS*, in the context of the Workpackage 2 activities.

References

Baader, F.; Calvanese, D.; McGuinness, D.; Nardi, D.; and Patel-Schneider, P. F., eds. 2003. Cambridge University Press.

Berardi, D.; Calvanese, D.; De Giacomo, G.; Lenzerini, M.; and Mecella, M. 2003a. A Foundational Vision of *e*-Services. In *Proc. of the CAiSE 2003 Workshop on Web Services, e-Business, and the Semantic Web (WES 2003)*.

Berardi, D.; Calvanese, D.; De Giacomo, G.; Lenzerini, M.; and Mecella, M. 2003b. Automatic Composition of *e*-Services that Export their Behavior. In Springer., ed., *Proc. of the 1st Int. Conf. on Service Oriented Computing (ICSOC 2003)*, volume 2910 of *LNCS*.

Berardi, D.; Calvanese, D.; De Giacomo, G.; Lenzerini, M.; and Mecella, M. 2003c. *e*-Service Composition by Description Logics Based Reasoning. In *Proc. of the 2003 International Workshop on Description Logics (DL'03)*.

Berardi, D.; Calvanese, D.; De Giacomo, G.; and Mecella, M. 2003d. Reasoning About Actions for *e*-Service Composition. In *Proc. of the ICAPS Workshop on Planning for Web Services (P4WS 2003)*.

Calvanese, D., and De Giacomo, G. 2003. Expressive description logics. In Baader et al. (2003). chapter 5, 178–218.

Harel, D.; Kozen, D.; and Tiuryn, J. 2000. *Dynamic Logic*. MIT Press.

Hull, R.; Benedikt, M.; Christophides, V.; and Su, J. 2003. *e*-Services: A Look Behind the Curtain. In *Proc. of PODS 2003*.

Papazoglou, M., and Georgakopoulos, D. 2003. Service Oriented Computing (special issue). *Comm. of the ACM* 46(10).