Composition of Services with Nondeterministic Observable Behavior

Daniela Berardi[1], Diego Calvanese[2],
Giuseppe De Giacomo[1], and Massimo Mecella[1]

[1] *Università di Roma "La Sapienza"*, Italy
lastname@dis.uniroma1.it

[2] *Free University of Bozen-Bolzano*, Italy
calvanese@inf.unibz.it

Abstract. In [3] we started studying an advanced form of service composition where available services were modeled as deterministic finite transition systems, describing the possible conversations they can have with clients, and where the client request was itself expressed as a (virtual) service making use of the same alphabet of actions. In [4] we extended our studies by considering the case in which the client request was loosen by allowing don't care nondeterminism in expressing the required target service. In the present paper we complete such a line of investigation, by considering the case in which the available services are only partially controllable and must be modeled as nondeterministic finite transition systems, possibly because of our lack of information on their exact behavior. Notably such services display a "devilish" form of nondeterminism, since we want to model the inability of the orchestrator to actually choose between different executions of the same action. We investigate how to automatically perform the synthesis of the composition under these circumstances.

1 Introduction

Service Oriented Computing (SOC) [1] is the computing paradigm that utilizes Web services (also called *e*-Services or, simply, services) as fundamental elements for realizing distributed applications/solutions. In particular, when no available service can satisfy client needs, (parts of) available services can be composed and orchestrated in order to satisfy such a request. In recent research [7, 8, 10, 15, 6, 9] a notion of "semantic service integration" is arising, especially to facilitate *automatic service composition* (but also discovery, etc.).

Among the various proposals, the one in [3, 4] distinguishes itself by considering also the process of the services. Specifically, the client is offered a set of virtual building blocks so that he can design complex services of interest in terms of these. The building blocks are *actions* described in an abstract and formal fashion; by making use of such virtual blocks the client can write its own service as a sort of high-level program, i.e., abstractly represented as a *deterministic finite transition system* (i.e., deterministic finite state machine)[1]. The virtual blocks are not be implemented directly, but made available through the system: the actual services that are available to the system are themselves be formally described in terms of deterministic finite transition systems built out of

[1] Transition systems here are used to formalize the possible conversations that a service can have with its clients – including the orchestrator in the case the service is involved in a composition – describing the possible interactions of the service.

such virtual blocks. Such a description can be considered as a sort of mapping from the concrete service to the virtual blocks of the integration system. The idea is to exploit the reverse of such a mapping to automatically get the client service request. In [3, 4], however, available services are modeled as deterministic transition systems because it is assumed that they are fully controllable by the orchestrator through action requests: an available service, by performing an action in a state, reaches exactly a single state.

In this paper we extend the approach of [3, 4] so as to address automatic composition synthesis when available services are not not fully controllable by the orchestrator. We model such a partial controllability by associating to available services (finite) transition systems that are *nondeterministic* (in a "devilish" sense, see later). Using nondeterminism we can naturally model services in which the result of each interaction with its client can not be foreseen. Just as an example, consider a service allowing to buy items by credit card; after invoking the operation, the service can be in a state `payment_OK`, accepting a payment, or in a different state `payment_refused`, if, e.g., the credit card is not valid. Considering that the transition system of the available service is in fact a mapping that describes the real service in terms of the actions of the community, it is natural to assume that although the orchestrator does not have full control on the available services, it has full observability: after executing the operation, it can observe the status in which the service is and therefore understand which transition, among the ones that are nondeterministically possible in the previous state, has been undertaken by the service[2]. The main contribution of our work is to show how one can synthesize a composition in this setting.

2 Services with Partially Controllable Behavior

Formally, we consider each *available service* as a *nondeterministic*[3] finite transition system $\mathcal{S} = (\Sigma, S, s_0, \delta, F)$ where Σ is a common alphabet of actions shared by all available services of a community, S is a finite set of states, $s_0 \in S$ is the single initial state, $\delta \subseteq S \times \Sigma \times S$ is the transition relation[4], and $F \subseteq S$ is the set of final states (i.e., states in which the computation may stop, but does not necessarily have to – see [3, 4]).

The client service request, as in [3], is expressed as a *target service*, which represents the service the client would like to interact with. Such a service is again modeled as finite transition system over the alphabet of the community, but this time a *deterministic* one, i.e., the transition relation is actually functional (there cannot be two distinct transitions with the same starting state and action). Notice that the target service is obviously deterministic because we assume that the client has full control on how to execute the service that he/she requires[5].

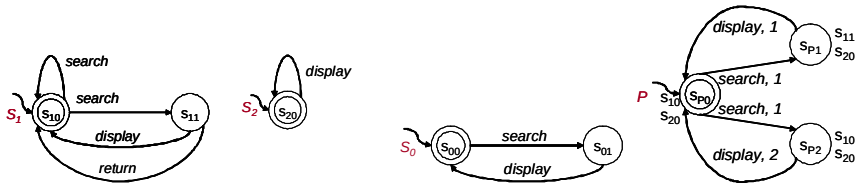
---

[2] The reader should observe that also the standard proposal WSDL 2.0 has a similar point of view: the same operation can have multiple output messages (the `out message` and various `outfault messages`), and the client observes how the service behaved by receiving a specific output message.

[3] Note that this kind of nondeterminism is of a *devilish* nature, so as to capture the idea that the orchestrator cannot fully control the available services.

[4] As usual, we call the Σ component of such triples, the *label of the transition*.

[5] In fact we could have a client request that is expressed as a nondeterministic transition system as in [4]. In this case, however, the nondeterminism has a *don't-care*, aka *angelic* nature.



(a) Available Services (b) Target Service (c) Composition

Fig. 1. Composition of nondeterministic services

Example 1. Figure 1(a) shows a community of services for getting information on books. The community includes two services: $\mathcal{S}_1$ that allows one to repeatedly *(i)* search the ISBN of a book given its title (`search`) then, *(ii)* in certain cases (e.g., if the record with cataloging data is currently accessible), it allows for displaying the cataloging data (such as editor information, year of publication, authors, copyrights, etc.) of the book with the selected ISBN (`display`), or *(iii)* simply returns without displaying information (`return`); $\mathcal{S}_2$ allows for repeatedly displaying cataloging data of books given the ISBN (`display`), without allowing researches. Figure 1(b) shows the target service $\mathcal{S}_0$: the client wants to have a service that allows him to search for a book ISBN given its title (`search`), and then display its cataloging data (`display`). Note that the client wants to display the cataloging data in any case and hence he/she can neither directly exploit $\mathcal{S}_1$ nor $\mathcal{S}_2$. □

Next, we need to clarify which are the basic capabilities of the orchestrator. In [3, 4], the orchestrator had only the ability of selecting one[6] of the services, and requiring it to execute an action. Here, we equip the orchestrator with a further ability: the orchestrator can query (at runtime) the current state of each available service. Technically such a capability is called *full observability* on the states of the available services. Although other choices are possible [15, 2], full observability is the natural choice in this context, since the transition system that each available service exposes to the community is specific to the community itself (indeed it is expressed using the common alphabet of actions of the community), and hence there is no reason to make its states partially unobservable: if details have to be hidden, this can be done directly within the transition system, possibly making use of nondeterminism.

3 Composition

We are now ready to define composition: an "orchestrator program" (indeed a skeleton specification) that the orchestrator has to execute in order to orchestrate the available services so as to offer to the client the target service. Let the available service be $\mathcal{S}_1, \ldots, \mathcal{S}_n$ each with $\mathcal{S}_i = (\Sigma, S_i, s_{i0}, \delta_i, F_i)$, and the target service $\mathcal{S}_0 = (\Sigma, S_0, s_{00}, \delta_0, F_0)$. A *history* is an alternating sequence of the form $h = (s_1^0, \ldots, s_n^0) \cdot a^1 \cdot (s_1^1, \ldots, s_n^1) \cdots a^\ell \cdot (s_1^\ell, \ldots, s_n^\ell)$ such that the following constraints hold:

[6] For simplicity we assume that the orchestrator selects only one service at each step, however our approach and results easily extend to the case where more services can be selected at each step.

– $s_i^0 = s_{i0}$ for $i \in \{1, \ldots, n\}$, i.e., all services start in their initial state;
– at each step k, for one i we have that $(s^k, a^{k+1}, s_i^{k+1}) \in \delta_i$, while for all $j \neq i$ we have that $s_j^{k+1} = s_j^k$, i.e., at each step of the history, only one of the service has made a transition (according to its transition relation), while the other ones have remained still.

An *orchestrator program* is a function $P : \mathcal{H} \times \Sigma \to \{1, \ldots, n, u\}$ that, given a history $h \in \mathcal{H}$ (where $\mathcal{H}$ is the set of all histories defined as above) and an action $a \in \Sigma$ to perform, returns the service (actually the service index) that will perform it. Observe that such a function may also return a special value u (for "undefined"). This is a technical convenience to make P a total function returning values even for histories that are not of interest or for actions that no service can perform after a given history.

Next, we define when an orchestrator program is a composition that realizes the target services. First, we observe that, since the target service is a deterministic transition system its behavior is completely characterized by the set of its traces, i.e., by the set of infinite sequences of actions that are faithful to its transitions, and of finite sequences that in addition lead to a final state[7]. Now, given a trace $t = a_1 \cdot a_2 \cdots$ of the target service, we say that an *orchestrator program P realizes the trace t* iff for each non-negative integer ℓ and for each history $h \in \mathcal{H}_t^\ell$, we have that $P(h, a_{\ell+1}) \neq u$ and $\mathcal{H}_t^{\ell+1}$ is nonempty, where the sets $\mathcal{H}_t^\ell$ are inductively defined as follows:

– $\mathcal{H}_t^0 = \{(s_{10}, \ldots, s_{n0})\}$
– $\mathcal{H}_t^{\ell+1}$ is the set of all histories such that, if $h \in \mathcal{H}_t^\ell$ and $P(h, a_{\ell+1}) = i$ (with $i \neq u$), then for all transitions $(s_i^\ell, a, s_i') \in \delta_i$ the history $h \cdot a_{\ell+1} \cdot (s_1^{\ell+1}, \ldots, s_n^{\ell+1})$, with $s_i^{\ell+1} = s_i'$, and $s_j^{\ell+1} = s_j^\ell$ for $j \neq i$, is in $\mathcal{H}_t^{\ell+1}$.

Moreover, if a trace is finite and ends after f actions, we have that all histories in $\mathcal{H}_t^f$ end with all services in a final state. Finally, we say that an *orchestrator program P realizes the target service $\mathcal{S}_0$*, if it realizes all its traces.

In order to understand the above definitions, let us observe that intuitively the orchestrator program realizes a trace if it can choose at every step an available service to perform the requested action. However, since when an available service executes an action it nondeterministically chooses what transition to actually perform, the orchestrator program has to play on the safe side and require that for each of the possible resulting states of the activated service, the orchestrator is able to continue with the execution of the next action. In addition, before ending a computation, available services need to be left in a final state, hence we have the additional requirement above for finite traces.

Example 1 (cont.) Figure 1(c) shows an orchestrator program P (in this case with finite states) for available services $\mathcal{S}_1$ and $\mathcal{S}_2$ in Figure 1(a), that realizes the target service $\mathcal{S}_0$ in Figure 1(b). Essentially, P behaves as follows: it repeatedly delegates to $\mathcal{S}_1$ the action `search` (notice that both transitions labeled with this actions are delegated to $\mathcal{S}_1$); then it checks the resulting state of $\mathcal{S}_1$ and, depending on this state, it delegates the action `display` to either $\mathcal{S}_1$ or $\mathcal{S}_2$. □

[7] Actually, the behavior captured by a transition system is typically identified with its execution tree, see [3]. However, since the target service has a deterministic transition system, the set of traces is sufficient, since one can immediately reconstruct the execution tree from it.

Observe also that the orchestrator program has to observe the states of the available services in order to decide which service to select next (for a given action requested by the target service). This makes such orchestrator programs akin to an advanced form of conditional plans studied in AI [12]. Observe also that, in the above definition we allow orchestrator program to have infinite states in general. But obviously it is of interest to understand in what circumstances composition may be realized through an orchestrator program that has only a finite number of state.

4 Composition Synthesis

It turns out that in spite of the additional complexity of dealing with nondeterminism, one can still devise a reduction from the problem of checking the existence of a composition to satisfiability in Propositional Dynamic Logic (PDL) [5] as in [3, 4]. The reduction is much more subtle in this case but still polynomial. As a result, we have that composition synthesis can be performed in EXPTIME. Moreover from each model of the resulting PDL formula one can directly extract an orchestrator program, and, considering the finite model property of PDL, this in turns implies that an orchestrator program that has only a finite number of states exists whenever a composition exists.

Actually, it comes quite as a surprise that in dealing with partial controllability one can still use a PDL encoding instead of directly working with automata on infinite trees [13]. And this finding is particularly welcome considering that certain operations on automata on infinite trees (e.g., the notorious Safra's complementation step) have proved to be almost impossible to implement in an efficient way. PDL satisfiability, instead shares the same basic algorithms behind the success of the description logics-based reasoning system used for OWL, and hence its use is quite promising.

5 Conclusion

In this paper we studied how to synthesize a composition to realize a client service request expressed as a target service a la [3, 4], in the case where available services are only partially controllable (modeled as devilish nondeterminism) but fully observable by the orchestrator. Such an approach to deal with nondeterministic available services can be extended in several directions. As an example, by introducing a set of *variables shared among the available services and the client* that encode some basic information that is exchanged between the services, and that the client acquires while executing the target service. Once we introduce shared variables, we can use them to guard transitions in both the target and the available services.

The result can be also easily extended to the case where the client request is expressed as a nondeterministic transition system as in [4]. Note that in this case the nondeterminism has a *don't-care*, aka *angelic*, nature: the client is not fully specifying the target service he/she requires, and allows some degree of freedom to the composer in providing him/her with one, by choosing among the nondeterministic transitions which one to actually implement. Such a form of nondeterminism can be still tackled through a reduction to satisfiability in PDL.

It should be noted that our approach, in which the orchestrator at each step sends an execution request to available services and these then send back to the orchestrator their

states, is a form of control that is communication intensive[8]. In fact, if communication is of concern, our model is too coarse. Indeed we should distinguish between actions that affect the state of affairs and messages for sending (either contents or control) information. Suggestions on tackling such a distinction are presented in [2].

Finally we want to stress that composition, especially in rich dynamic settings as those studied in this paper, is essentially a form of (reactive) program synthesis, and tight relationships exist with the literature on that field [11, 14, 16]. Although that literature often does not offer off-the-shelf results for composition, it certainly offers techniques and general approaches that can be profitably used to tackle subtle issues, as, for example, partial observability, which becomes an issue when the distinction between actions and messages is taken into account.

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services. Concepts, Architectures and Applications*. Springer, 2004.
2. D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella. Automatic composition of transition-based semantic web services with messaging. In *Proc. of VLDB 2005*, 2005.
3. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *Proc. of ICSOC 2003*.
4. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Synthesis of underspecified composite e-Services based on automated reasoning. In *Proc. of ICSOC 2004*.
5. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, 2000.
6. R. Hull and J. Su. Tools for design of composite web services. In *Proc. of ACM SIGMOD*, pages 958–961, 2004.
7. U. Kuter, E. Sirin, D. Nau, B. Parsia, , and J. Hendler. Information gathering during planning for web service composition. In *Proc. of Workshop on Planning and Scheduling for Web and Grid Services*, 2004.
8. S. A. McIlraith and T. C. Son. Adapting Golog for composition of semantic web services. In *Proc. of KR 2002*, pages 482–496, 2002.
9. B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing web services on the semantic web. *VLDB Journal*, 12(4):333–351, 2003.
10. M. Michalowski, J. L. Ambite, C. A. Knoblock, S. Minton, S. Thakkar, and R. Tuchinda. Retrieving and semantically integrating heterogeneous data from the web. *IEEE Intelligent Systems*, 19(3):72–79, 2004.
11. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. of POPL'89*, pages 179–190, 1989.
12. J. Rintanen. Complexity of planning with partial observability. In *Proc. of the 14th Int. Conf. on Automated Planning and Scheduling (ICAPS 2004)*, pages 345–354, 2004.
13. W. Thomas. Languages, automata, and logic. In *Handbook of Formal Language Theory*, volume III, pages 389–455. 1997.
14. W. Thomas. Infinite games and verification. In *Proc. of CAV 2002*, volume 2404 of *LNCS*, pages 58–64. Springer, 2002.
15. P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *Proc. of ISWC 2004*, volume 3298 of *LNCS*, pages 380–394. Springer, 2004.
16. M. Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In *Proc. of CAV'95*, volume 939 of *LNCS*, pages 267–292. Springer, 1995.

[8] Actually we had essentially the same amount of control communication in [3, 4]: indeed even if states were not sent back to the orchestrator, at least some feedback to signaling the readiness to accept further commands should have been sent back.