Chapter 8
Automated Service Composition Based on Behaviors: The Roman Model

Giuseppe De Giacomo, Massimo Mecella and Fabio Patrizi

Abstract During the last years, many approaches have been proposed in order to address the issue of automated service composition. In this chapter, we discuss the so-called “Roman model”, in which services are abstracted as transition systems and the objective is to obtain a composite service that preserves a desired interaction, expressed as a (virtual) target service. We will also outline its deployment in the challenging applications of smart houses, i.e., buildings pervasively equipped with sensors and actuators making their functionalities available according to the service-oriented paradigm.

8.1 Introduction

Services are software artifacts, possibly distributed and built on top of different technologies, that export a description of themselves, are accessible to external clients and communicate through a commonly known, standard interface which enables interoperability. More in general, Service Oriented Computing (SOC) is a computing paradigm whose basic elements are services, that can be used as building blocks to devise other services. A classical example of such a paradigm is provided by *Web services*, i.e., applications published over the Internet and self-described, usually built by different companies and relying on different technologies, which share a same communication protocol, namely SOAP. For instance, many online travel

G. De Giacomo · M. Mecella (✉) · F. Patrizi
Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti,
Sapienza Università di Roma, Via Ariosto 25, 00185 Roma, Italy
e-mail: degiacomo@dis.uniroma1.it

M. Mecella
e-mail: mecella@dis.uniroma1.it

F. Patrizi
e-mail: patrizi@dis.uniroma1.it

A. Bouguettaya et al. (eds.), *Web Services Foundations*,
DOI: 10.1007/978-1-4614-7518-7_8,

agencies integrate different Web services offered by hotels, airlines, restaurants, etc., to provide final users with a complete service, combining all functionalities of its basic components. No constraints are required over the *internal* structure of each Web service, but they are all required to be published, compliant with the same communication protocol and to export a description of their interface, so as to facilitate access and communication.

Abstracting from this example, services can be thought as generic programs, publicly available and wrapped so as to mutually interact and communicate over a common platform. As such, the SOC paradigm makes easier code re-use and extension, as, in a sense, each service is interpreted as a procedure/method in programming languages and, thus, a set of services as a sort of *programming library*. This similarity can be taken as the basis of *service composition*: as exactly as in a programming language procedures/methods are combined to produce more complex procedures/methods, so services can be combined to build more complex services.

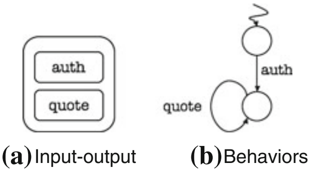
This chapter focuses on *automated service composition*, that is, the problem of automatically combining a set of available services, so as to meet a desired specification. To this end, we start from the classical architecture for Web services. The parties typically involved include a client, that can be a service itself, the *directory*, and a set of *service providers*. The directory is a central, publicly available registry storing service descriptions which allow clients to search for some desired service; service providers are organizations, typically companies, that publish actual running services, advertised in registries. A typical session is as follows: (i) a client searches for a desired service, e.g., weather forecasting, in a directory; (ii) if the service is found, the client is redirected to the provider that deploys the service; (iii) the client contacts the desired service and interacts with it, according to its needs. This simple scenario is already sufficient to raises two classical questions in SOC: (i) *how to describe services?* (ii) *what if the desired service is not found?* The first one concerns *service modeling*, i.e., the definition of a suitable abstraction of services, able to capture aspects that can be relevant to clients; the second one raises the problem of finding a constructive alternative to the trivial answer: "the request cannot be fulfilled". As one may expect, there exist many reasonable, correct answers to them. In this work we discuss both problems. We first present a model, sometime referred to as the "Roman Model", that substantially enriches existing ones, by providing an abstraction of the *conversations* a service can carry on with clients; then, on top of this model, we describe a technique for *building* a solution that fulfills a client request by suitably combining the available services. In addition, we show that such techniques is in fact *best one can do*, in the sense of returning the most general solution, while being optimal with respect to worst-case time complexity.

8.1.1 Modeling Behaviors

In the literature, several approaches to service modeling have been proposed. Rather than actual languages widely used to describe Web services, such as WSDL, we focus on their conceptual model. A WSDL description exports a *functional* specification of a service, that is, from an abstract standpoint, the set of operations provided by the service, along with the corresponding format of messages exchanged. We can say that WSDL has an underlying *atomic* conceptual model, specified in terms of input-output requirements. For instance, a service providing stock quotes of some market can be successfully described this way, with a single operation that returns the list of quotes. However, when more complex specifications need to be exported, it shows severe limitations. For instance, let us consider the same Web service for stock quotes and assume that it provides quotations only to authenticated clients. In an input-output approach, one would describe two operations, `auth` and `quote`, as well as the respective data format necessary for interaction. Unfortunately, the input-output approach does not allow for *conversation specification*, i.e., for putting constraints on the order that operations should be executed in. A very natural constraint would be, e.g., requiring clients to authenticate before requesting quotes. Observe also that cases may exist where two services export a same set of operations but allow different execution sequences. Since this last constraint is not captured by input-output approaches, such services would appear to clients as the same. In a word, atomic conceptual models export services' *interface* but not their *behavior*.

The need for a *behavioral* description of services has been already recognized in the literature, e.g., [3], yet the community suffers from a lack of standard languages for this purpose. In this work, we present the so-called *Roman Model* (as named by [23]), originally introduced in [6], and oriented to describe all *conversations* supported by services, that includes (in its various variants) relevant features, such as nondeterminism and shared memory.

In our model, services export their behavioral features by means of a language that represents transition systems, i.e., Kripke structures whose transitions are labeled by service's operations, under the assumption that each legal run of the system corresponds to a conversation supported by the service. To clarify this, consider Fig. 8.1. The former is a graphical representation of an input–output description of the stock quote service with authentication, which provides information about operations that can be requested; the latter is a behavioral representation of the same service, providing more information: indeed, it tells clients that they (i) must authenticate before requesting a `quote` operation and, then, (ii) may request any number of quotes. Of course, more sophisticated examples do exist, where several operations, even nondeterministic, can be executed in a state, with nondeterminism modeling partial knowledge about service's internal logic. Also, there are settings relying on the same approach, where operations have parameters and are able to exchange data with other clients and even with an underlying database (cf. [5]).



(a) Input-output **(b)** Behaviors

Fig. 8.1 Service descriptions. **a** Input–output. **b** Behaviors

A first advantage brought by such a model is its *generality* with respect to service integration, in the sense that it is abstract enough to serve as conceptual model for several classes of scenarios. As an example, it can be used to model Web service applications as well as multi-agent system ones. As a consequence, results obtained on this model are also relevant to areas different from SOC. Second, from the SOC viewpoint, it provides a behavioral, stateful, service representation, which allows for describing those inter-operation constraints that current languages, e.g., WSDL, do not capture. We remark the importance of such a feature in a perspective of composition automatization: indeed, composition engines are intended to replace human operators, who compose services based on their informal description, often provided in natural language, which include behavioral information. Importantly, when dealing with a behavioral model, we can look at services as high-level descriptions of software artifacts. Indeed, they are characterized by states and state transitions triggered by inputs, which, specifically, represent requested operations. This interpretation suggests, hence, to see service (possibly finite) runs as computation fragments, that can be suitably combined to generate more complex services.

8.1.2 Composing Services

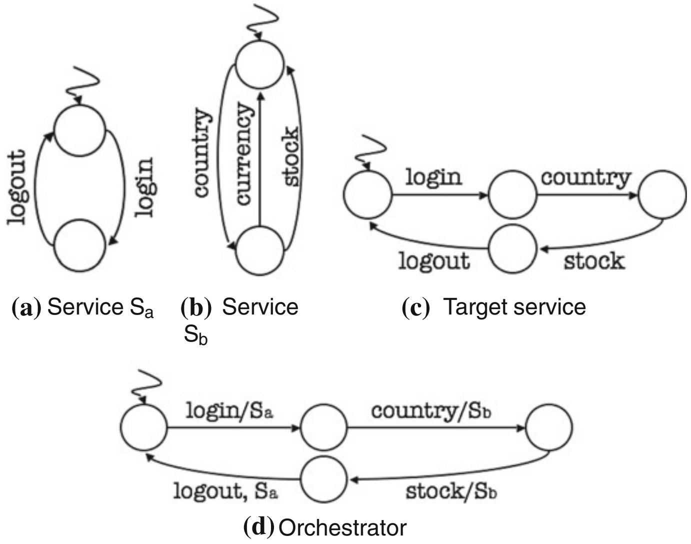
Many works exist which deal with automated composition of services (see Sect. 8.2 for a survey). Our problem can be informally stated as follows:

> Consider a set of available services, a.k.a. *community*, and an additional *target service*, all exporting their conversational behaviors. Is it possible to coordinate the available services so to support, at execution time, all conversations supported by the target service?

In other words, the problem amounts to realize a (virtual) target service, by resorting only to (actual) available services. Obviously, how services are combined in the practice depends on the exported behavioral models. To see how this can be done, consider the following example.

Example 8.1 Figure 8.2 shows a service composition problem instance in the Roman Model, which includes two available services, represented in subfigures (a) and (b), and a target one, in subfigure (c). The one in subfigure (a), say S_a, provides login/logout capabilities, allowing a client to be authenticated and to close an authenticated session, whereas the one in subfigure (b), say S_b, provides market stock quotes from all over the world. Clients willing to interact with S_b are, first, required to input the market country of their interest and, then, are allowed to request either stock quotes or currency rates (versus, e.g., euro and dollar) for that market. As for the target service, say T, it provides stock quotes of a selected market only to authenticated clients. Specifically, clients of such a service need first to login, then to select a market country, then are allowed to request quotes and, finally, to logout.

As we said, target services are *virtual*, that is, only their specification exists, whereas their implementation is missing. However, it is easily seen that, by resorting



(a) Service S_a **(b)** Service S_b **(c)** Target service

(d) Orchestrator

Fig. 8.2 A service composition example in the Roman Model. **a** Service S_a. **b** Service S_a. **c** Target service. **d** Orchestrator

to available services, this example's target service can be built. Indeed, it is enough *delegating* login/logout operations to S_a and country selection and stock requests to S_b. Observe that the target service not only provides a set of operations, but imposes a set of constraints over their executions, e.g., `stock` can be requested only after `country` has been executed. Since, on their side, also available service operations are subject to such a kind of constraints, when a target service is to be realized, they must be met. For instance, had not T required operation `country` be executed before `stock`, it would be not realizable, as S_b is the only service that provides `stock` and it requires `country` to be executed first.

The composition can be realized by a machine which, on one side, receives client's operation requests and, on the other side, forwards them to an appropriate available service which executes it and, consequently, changes its state, where a new set of operations becomes available. Such a machine, similar to a Mealy machine but that can be, in general, infinite-state, is called *orchestrator*. A possible orchestrator is shown in subfigure (d). Each state of the machine corresponds to a state of the target service and each transition is labeled by a pair of the form $operation/service$, with an intuitive semantics: the requested operation is assigned to the output service. For instance, operation `login` is delegated to service S_a.

The example above shows how the existence of temporal constraints among operation executions makes the problem non trivial: each time an operation is to be delegated to some available service, one needs to check whether all constraints are fulfilled, i.e., whether the service chosen for delegation is in a state where the operation is actually executable. This makes the orchestrator construction an hard task:

in the Roman Model, the service composition problem is shown to be EXPTIME-complete [6, 32].

More complex scenarios can be considered. For instance, nondeterministic available services are also conceivable, where nondeterminism over operation execution represents partial knowledge about service's internal logic. Also, one could think of services communicating through a common blackboard or even exchanging data. All these scenarios require different notions of composition and, hence, different kind of orchestrators.

8.1.3 The History of the Roman Model

The specific composition problem has been tackled with different techniques, starting by exploiting a reduction to satisfiability in a well-known logic of programs, namely PDL [6, 8, 9].[1] Notably, Logics of Programs are tightly related to Description Logics, for which highly optimized satisfiability checkers exist (e.g., RacerPro, Pellet, FACT, etc.). This framework has been then extended to consider interesting variants, e.g.: forms of target service loose specifications [7], trust-aware services [13], distributed orchestrators [35], shared environments [18], data-aware services [5].

More recently, another approach has been proposed based on computing compositions by exploiting (variants of) the formal notion of simulation [10, 34]. Interestingly, through this, the case where the state of services is only partially observable has been also addressed [16]. The solution technique directly appeals to techniques for Linear Time Logic (LTL) synthesis, to model-check a game structure representing a so-called *safety-game*. Since this can be realized in practice on top of symbolic model checking technologies, the approach gained a high level of scalability, and has been effectively realized in the context of an EU research project (see Sect. 8.4). In the following we will focus on this latter approach.

8.2 State-of-the-art on Automated Service Composition

In order to discuss *automated* service composition, and compare different approaches, we introduce here a sort of conceptual framework for "semantic service integration", that is constituted by the following elements[2]: (i) the *community ontology*, which represents the common understanding on an agreed upon reference semantics between

[1] The reader should note that [6] has been historically one of the most cited papers in the automated service composition field, cf. more than 390 citations according to Google Scholar—September 2012. The same for [5] (cf. more than 250 citations).

[2] Such a framework is inspired by the research on "semantic data integration" [27]. Obviously that research has dealt with data (i.e., static aspects) and not with computations (i.e., dynamic aspects) that are of interest in composition of services. Still many notions and insights developed in that field may have a deep impact in service composition. An example is the distinction that we make later

the services,[3] concerning the meaning of the offered operations, the semantics of the data flowing through the service operations, etc; (ii) the set of *available services*, which are the actual Web services available to the community; (ii) the *mapping* for the available services to the community ontology, which expresses how services expose their behavior in terms of the community ontology; and (iv) the *client service request*, to be expressed by using the community ontology.

In general, the community ontology comprises several aspects: on one side, it describes the semantics of the information managed by the services, through appropriate semantic standards and languages; on the other side, it should consider also some specification of the service behaviors, on possible constraints and dependencies between different service operations, not limited solely to pre- and post-conditions, but considering also the process of the service. In building such a "semantic service integration" system, two general approaches can be followed. (i) In the *service-tailored* approach, the community ontology is built mainly taking into account the available services, by suitably reconciling them; indeed the available services are directly mapped as elements of the community ontology, and the service request is composed by directly applying the mappings for accessing concrete computations. (ii) Conversely in the *client-tailored* one, the community ontology is built mainly taking into account the client, independently from the services available; they are described (i.e., mapped) by using the community ontology, and the service request is composed by reversing these mappings for accessing concrete computations.

In fact, most of the research on automated service composition has adopted a service-tailored approach. For example, the works based on Planning in AI (e.g., [38, 40, 33]) consider services as atomic actions—only I/O behavior is modeled, and the community ontology is constituted by propositions/formulas (facts that are known to be true) and actions (which change the truth-value of the propositions); available services are mapped into the community ontology as atomic actions with pre- and post-conditions. In order to render a service as an atomic action, the atomic actions, as well as the propositions for pre- and post-conditions, must be carefully chosen by analyzing the available services, thus resulting in a service-tailored approach.

Other works (e.g., Papazoglou's et al. [39], Bouguettaya et al. [30], Sheth et al. [12]) have essentially considered available services as atomic actions characterized by the I/O behavior and possibly effects. But differently from those based on planning, instead of concentrating on the automatic composition, they have focused more on modeling issues and automated discovery of services described making use of rich ontologies.

Also the work of McIlraith et al. [29] can be classified as service-tailored: services are seen as (possibly infinite) transition systems, the common ontology is a Situation Calculus Theory (therefore is semantically very rich) and service names, and each

between "service-tailored" and "client-tailored" service integration systems, which roughly mimic the distinction between Global As View (GAV) and Local As View (LAV) in data integration.

[3] Note that many scenarios of cooperative information systems, e.g., *e*-Government or *e*-Business, consider preliminary agreements on underlying ontologies, yet yielding a high degree of dynamism and flexibility.

service name in the common ontology is mapped to a service seen as a procedure in Golog/Congolog Situation Calculus; the client service request is a Golog/Congolog program having service names as atomic actions with the understatement that it specifies acceptable sequences of actions for the client (as in planning) and not a transition system that the client wants to realize.

Finally, the work by Hull et al. [11] describes a setting where services are expressed in terms of atomic actions (communications) that they can perform, and channels linking them with other services. The aim of the composition is to refine the behavior of each service so that the conversations realized by the overall system satisfy a given goal (dynamic property) expressed as a formula in LTL. Although possibly more on choreography synthesis than on composition of the form discussed here, we can still consider it a service-tailored approach, since there is no effort in hiding the service details from the client that specifies the goal formula.

Much less research has been done following a client-tailored approach, but some remarkable exceptions should be mentioned: the work of Knoblock et al. [31] is basically a data integration approach, i.e., the community ontology is the global schema of an integrated data system, the available services are essentially data sources whose contents is mapped as views over the global schema, and the client request is basically a parameterized query over such a schema; therefore the approach is client-tailored, but neither the ontology nor mappings consider service behavior at all.

The work of Traverso et al. [33] can be classified also as client-tailored: services are seen as (finite) transition systems, the common ontology is a set of atomic actions and propositions, as in Planning; a service is mapped to the community ontology as a transition system using the alphabet of the community and defining how transitions affect the propositions, and the client service request asks for a sequence of actions to achieve GOAL1 (main computation), with guarantees that upon failure GOAL2 is reached (exception handling).

Finally, the line of research taken in [6–9], but also in [21], has the dynamic behavior of services at the center of its investigation. In order to study the impact of such dynamics on automatic composition, all these works make simplifying assumptions on the community ontology, which essentially becomes an alphabet of actions. Still the notion of community ontology is present, and in fact all these works adopt a client-tailored approach. A fundamental issue that arises is whether such rich descriptions of the dynamic behavior of the services can be combined with rich (non propositional) descriptions of the information exchanged by the services, while keeping automated composition feasible. The first results on this issue were reported in [5], where available services that operate on a shared world description (in a form of a database) are considered. Such services can either operate on the world through some atomic processes as in OWL-S, or exchange information through messages. While the available services themselves are with finite states, the world description is not. Under suitable assumptions on how the world can be queried and modified, decidability of service composition is shown. Interestingly [5] shows that even if the available services can be modeled as deterministic transition systems, the presence

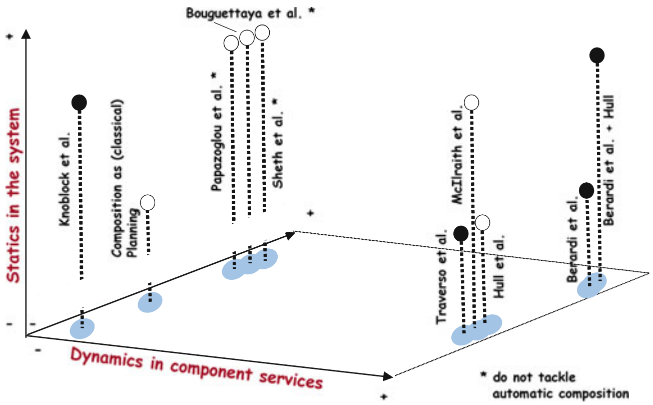


Fig. 8.3 Comparison of the various approaches

of a world description whose state is not known at composition time, requires dealing with nondeterminism.

Figure 8.3 summarizes, on the basis of the previous discussion, the considered works. The three axis represent the levels of detail according to which the community ontology and the mappings and the client request can be modeled. Namely, (i) *statics in the system* represents how fine grained is the modeling of the static semantics (i.e., ontologies of data and/or services, inputs and outputs, alphabet of actions, etc.); (ii) *dynamics in component services* represents how fine grained is the modeling of the processes and behavioral features of the services (only atomic actions, transition systems, etc.); and (iii) *dynamics in client service request* represents how fine grained is the modeling of the process required by the client, varying from a single step (as in the case of services consisting essentially of queries over a data integration system) to a (set of) sequential steps, to a (set of) conditional steps, to including loops, up to running under the full control of the client (as in our approach). *Black/white lollipops* represent service-tailored (white) versus client-tailored (black) approaches.

Finally, in the last years, many works (e.g., [1, 19, 26, 36] consider how to perform composition by taking into account Quality-of-Service (QoS) of the composite and component services. Moreover, some works consider non classical techniques (e.g., [37] adopts learning approaches) for solving the composition problem.

8.3 The Roman Approach

The approach to service composition described here falls into the client-tailored class. Its distinguishing features can be summarized as follows:

- The available services are grouped together into a so-called *community* (many other approaches, e.g., [4], consider the notion of community as central in the composition process).

- Services in a community share a common set of actions Σ, the *actions of the community*.
- Actions in Σ denote (possibly complex) interactions between service and clients. As a result of an interaction the client may acquire new information (not necessarily modeled in the description) that may affect the next interaction.
- The behavior of each available service is described in terms of a *finite transition system* that uses only actions from Σ.
- The desired service, called the *target service*, is itself described as a finite, deterministic transition system that uses actions from Σ. Determinism here captures the absence of uncertainty over the desired behavior.
- The orchestrator has the ability of scheduling services on a *step-by-step* basis.

In this approach, the *composition synthesis* task consists in synthesizing an *orchestrator* able to coordinate the community services so as to mimic the behavior of the target service. Differently put, the behavior obtained by coordinating the services should present no differences, from the client perspective, with the target service.

To describe this setting in terms of the framework previously discussed, we identify the following correspondences:

- the community ontology is simply Σ;
- the available services are the actual services in the community;
- the mapping from the available services to the community ontology is represented by the transition systems that describe the available services (built from community actions);
- the client request is the target service (again, built from community actions).

In [6, 8], the simple case where available services are modeled as deterministic finite transition systems is addressed, while in [9], (diabolic) nondeterminism has been introduced, to account for those situations where the orchestrator cannot control the outcome of interactions. The presence of nondeterministic conversations stems naturally when services offer interactions with an unforeseeable result. For instance consider a service that allows one to purchase items with a credit card. After obtaining the credit card details, the service interacts with the bank, to request payment authorization. If it is granted, the service offers the client the option to confirm the payment, while in case of denial the service offers the possibility of entering the details again. As it can be seen, the next options made available to clients depend on the outcome of the authorization request, which, from the outside perspective, is nondeterministic. As a result, the service itself is nondeterministic, from the perspective of its clients. Notice that after an interaction has taken place, its result becomes observable, that is, clients can know the state that the service has moved to. This feature can thus be exploited by the orchestrator (which is in fact a particular client), that can observe the current state of the available services and choose how to carry on a certain task.[4]

[4] The reader should observe that also the standard proposal WSDL 2.0 adopts a similar approach: an operation can have multiple output messages (the `out message` and various `outfault`

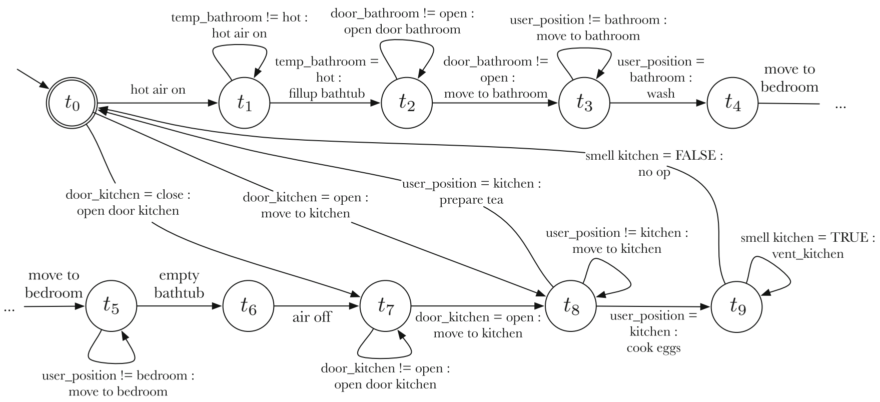


Fig. 8.4 Target service for the smart-house scenario

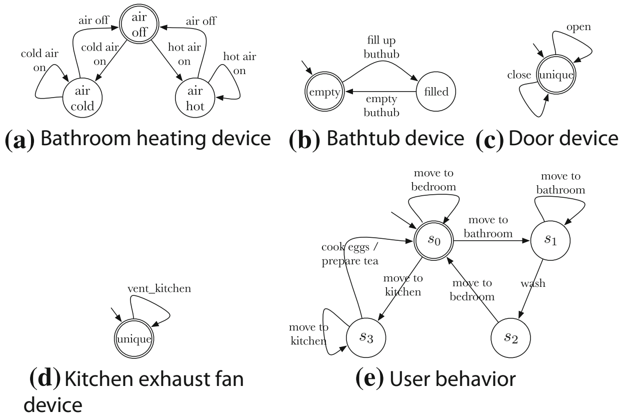
In the following, we present some technical details of the Roman approach, by considering non-deterministic services and the presence of data. In doing so, we use a running example from the context of smart houses, an interesting application scenario that our approach has been fully implemented in, proving effective. In this context, the composition goal is to generate an orchestrator that realizes some desired *routines* requested by the user, i.e., predefined sequences of operations that the house is intended to execute by suitably exploiting some devices (considered as services). For instance, a typical request issued in the morning could require heating the bathroom, lifting the shutters, and preparing a coffee, while at night, a user might request closing the shutters, locking the door, and switching the lights off.

8.3.1 The Framework

Technically, the behavior of services and the state of the house, called *environment*, are abstracted as finite-state transition systems. In details: each service is represented as a nondeterministic transition system (to model partial controllability); the user request, called *target*, is represented as a deterministic transition system (to model full controllability); and the environment is represented as a nondeterministic transition system (to model partial predictability). The state of the environment is assumed fully observable by all services, including the target. Our ultimate goal is to *simulate* the target by suitably delegating actions to the available services, as they are requested by the client.

For an example consider Fig. 8.4, which shows a fragment of a target behavior for the smart house scenario. It captures some requests typically issued by a user in

`messages`), and the client observes how the service behaved only after receiving a specific output message.



(a) Bathroom heating device (b) Bathtub device (c) Door device

(d) Kitchen exhaust fan device (e) User behavior

Fig. 8.5 Service community for the smart-house scenario. **a** Bathroom heating device. **b** Bathtub device. **c** Door device. **d** Kitchen exhaust fan device. **e** User behavior

the morning: having a shower and breakfast. States t_1, t_2, and t_3 contain the requests for heating the bathroom ("hot air on"), filling up the bathtub, opening the bathroom door, etc., that is, all the actions necessary to have a shower; the remaining states correspond to the actions to execute in order to have a breakfast ready. Checking whether these requests can be fulfilled in the proper order and, if so, which devices can be used to perform the actions, is exactly the objective of the synthesis task.

Figure 8.5 shows the set of available services, i.e., the *community*, for the same scenario. Notice that also the user is represented as a service. This is because users can in general execute actions that contribute to the realization of a target. Obviously, when this is not desired, a user can be simply excluded themselves from the community. For the environment, we consider the following state variables, with respective domain:

- $temp\_bathroom : \{warm, hot, cold\}$;
- $user\_position : \{bedroom, bathroom, kitchen\}$;
- $door\_bathroom : \{closed, open\}$;
- $door\_kitchen : \{closed, open\}$;
- $smell\_kitchen : boolean$;

in which every state variable assignment corresponds to a different environment state.

8.3.1.1 Environment and Behaviors

Formally, we have a shared nondeterministic, fully observable environment, which provides an abstract account of action preconditions and effects, and a mean of communication among services. In details, an <u>*environment*</u> is a tuple $\mathcal{E} = \langle \mathcal{A}, E, e_0, \rho \rangle$, where:

- $\mathcal{A}$ is a finite set of shared actions;
- E is the finite set of environment states;
- $e_0 \in E$ is the initial state;
- $\rho \subseteq E \times \mathcal{A} \times E$ is the transition relation among states: $\langle e, a, e' \rangle \in \rho$, or $e \xrightarrow{a} e'$ in $\mathcal{E}$, denotes that action a performed in state e may lead the environment to a successor state e'.

Services stand for the interface that available devices expose. At each step, a service offers a set of executable actions that can be chosen by the client. The client selects one, the service executes it, and a new step starts. In general service executions affect the environment (cf. above), hence they are equipped with the ability to test conditions (i.e., guards) on the environment, when needed. A (service) *behavior* over an environment $\mathcal{E} = \langle \mathcal{A}, E, e_0, \rho \rangle$ is a tuple $\mathcal{B} = \langle B, b_0, G, \varrho \rangle$, where:

- B is the finite set of behavior states;
- $b_0 \in B$ is the initial state;
- G is a set of *guards*, that is, boolean functions $g : E \mapsto \{\texttt{true}, \texttt{false}\}$;
- $\delta \subseteq B \times G \times \mathcal{A} \times B$ is the behavior transition relation, where $\langle b, g, a, b' \rangle \in \varrho$, or $b \xrightarrow{g,a} b'$ in $\mathcal{B}$, denotes that action a executed in state b, when the environment is in a state e such that $g(e) = \texttt{true}$, may lead the behavior to state b'.

The target in Fig. 8.4 has guarded actions, e.g., $temp\_bathroom\ != hot : hot\ air\ on$, meaning that action $hot\ air\ on$ can be requested only if the environment is in a state where $temp\_bathroom\ != hot$ holds. We then decouple the state of physical device from that of the house, coping with unpredictable situations. Suppose the plan is running and the orchestration module instructs the bath heating device to switch the hot air on, while a tenant is switching the air off when leaving the bathroom: thanks to guards, the plan does not progress until hot temperature is reached. In other words, the fact that the bath heating device is in state $hot\ air\ on$ does not yield that the bathroom temperature is actually hot.

As discussed, behaviors are *nondeterministic*. That is, given a state and an action, there may be several transitions whose guards evaluate to true. We say that a behavior $\mathcal{B}$ over $\mathcal{E}$ is *deterministic* if for no behavior state $b \in B$ and no environment state $e \in E$ there exist two transitions $b \xrightarrow{g_1,a} b'$ and $b \xrightarrow{g_2,a} b''$ in $\mathcal{B}$ such that $b' \neq b''$ and $g_1(e) = g_2(e) = \texttt{true}$. Obviously, given a state of a deterministic behavior and a legal action, we know exactly *the* next behavior state, while this is not the case for nondeterministic behaviors. Thus, we say that the former are *fully controllable* while the latter are only *partially controllable*.

Finally, we define a *system* $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ as an environment $\mathcal{E}$ and n predefined *available behaviors* $\mathcal{B}_i$ over $\mathcal{E}$. A *target behavior* is a *deterministic* behavior over $\mathcal{E}$ that represents the fully controllable desired behavior to be obtained through the available behaviors.

Let us analyze the target of Fig. 8.4. The transition system represents the actions that a user may ask at each moment in time. In state t_0, the initial one, the user can make a choice: either to have a shower and then to have breakfast, or to have a breakfast only. In the first case he asks for action $hot\ air\ on$, otherwise, he can $move\ to\ kitchen$

only if the kitchen door is open, or, if not, ask for $open\ door\ kitchen$. Let us suppose he decides to have a shower. In state t_1 he may request to $fillup\ bathtub$ (guarded action) only if the bathroom temperature is reasonably hot. Then he may ask to $open\ door\ bathroom$ (guarded) and only when it is opened he can move to the bathroom, wash and go back to the bedroom. Unless the system is sure that the user is back in the bedroom, the bathtub cannot be emptied, and the hot air cannot be switched off in the bathroom. After having a shower, the user is supposed to have breakfast. So, when the kitchen door is open, he can decide to either prepare a tea or cook eggs. In the latter case, the house system should vent the kitchen until the smell is gone. After these activities, the target returns in its starting state, allowing the tenants to repeat infinitely many times the same sequences of actions.

8.3.2 Enacted Behaviors

To show how the composition task is automatically carried out, we introduce some intermediate notions. Given a behavior $\mathcal{B}$ over $\mathcal{E}$, the enacted behavior of $\mathcal{B}$ over $\mathcal{E}$ is the tuple $\mathcal{T}_{\mathcal{B}} = \langle S, \mathcal{A}, s_0, \delta \rangle$, where:

- $S = B \times E$ is the (finite) set of $\mathcal{T}_{\mathcal{B}}$ states—given a state $s = \langle b, e \rangle$, we denote b by $beh(s)$ and e by $env(s)$;
- $\mathcal{A}$ is the set of actions in $\mathcal{E}$;
- $s_0 \in S$, with $beh(s_0) = b_0$ and $env(s_0) = e_0$, is the initial state of $\mathcal{T}_{\mathcal{B}}$;
- $\delta \subseteq S \times \mathcal{A} \times S$ is the enacted transition relation, where $\langle s, a, s' \rangle \in \delta$, or $s \xrightarrow{a} s'$ in $\mathcal{T}_{\mathcal{B}}$, iff: (i) $env(s) \xrightarrow{a} env(s')$ in $\mathcal{E}$; and (ii) $beh(s) \xrightarrow{g,a} beh(s')$ in $\mathcal{B}$, with $g(env(s)) = \mathtt{true}$ for some $g \in G$.

Technically, $\mathcal{T}_{\mathcal{B}}$ is the synchronous product of the behavior and the environment, and represents all the possible executions obtainable by executing $\mathcal{B}$, once guards are evaluated and actions are performed on $\mathcal{E}$. Observe that both the environment and the behavior are possible sources of nondeterminism for an enacted behavior.

All available behaviors in a system act concurrently, in an interleaved fashion, in the same environment. For simplicity, we assume that behaviors are *asynchronous*, that is, exactly one moves at each step.[5] The behavior emerging from the joint execution of all the available behaviors on an environment is referred to as the *enacted system behavior*. Let $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ be a system, where $\mathcal{E} = \langle \mathcal{A}, E, e_0, \rho \rangle$ and $\mathcal{B}_i = \langle B_i, b_{i0}, G_i, \varrho_i \rangle$ ($i \in \{1, \ldots, n\}$). The enacted system behavior of $\mathcal{S}$ is the tuple $\mathcal{T}_{\mathcal{S}} = \langle S_{\mathcal{S}}, \mathcal{A}, \{1, \ldots, n\}, s_{\mathcal{S}0}, \delta_{\mathcal{S}} \rangle$, where:

- $S_{\mathcal{S}} = B_1 \times \cdots \times B_n \times E$ is the finite set of $\mathcal{T}_{\mathcal{S}}$ states; when $s_{\mathcal{S}} = \langle b_1, \ldots, b_n, e \rangle$, we denote b_i by $beh_i(s_{\mathcal{S}})$, for $i \in \{1, \ldots, n\}$, and e by $env(s_{\mathcal{S}})$;

[5] In fact, it is possible to extend the approach and results presented here, to the case in which at each step more than one available behaviors acts as in [35].

- $s_{\mathcal{S}0} \in S_{\mathcal{S}}$ with $beh_i(s_{\mathcal{S}0}) = b_{i0}$, for $i \in \{1, \ldots, n\}$, and $env(s_{\mathcal{S}0}) = e_0$, is the initial state of $\mathcal{T}_{\mathcal{S}}$;
- $\delta_{\mathcal{S}} \subseteq S_{\mathcal{S}} \times \mathcal{A} \times \{1, \ldots, n\} \times S_{\mathcal{S}}$ is the $\mathcal{T}_{\mathcal{S}}$ transition relation, where $\langle s_{\mathcal{S}}, a, k, s'_{\mathcal{S}}\rangle \in \delta_{\mathcal{S}}$, or $s_{\mathcal{S}} \xrightarrow{a,k} s'_{\mathcal{S}}$ in $\mathcal{T}_{\mathcal{S}}$, iff:
  - $env(s_{\mathcal{S}}) \xrightarrow{a} env(s'_{\mathcal{S}})$ in $\mathcal{E}$;
  - $beh_k(s_{\mathcal{S}}) \xrightarrow{g,a} beh_k(s'_{\mathcal{S}})$ in $\mathcal{B}_k$, with $g(env(s_{\mathcal{S}})) = \texttt{true}$, for some $g \in G_k$; and
  - $beh_i(s_{\mathcal{S}}) = beh_i(s'_{\mathcal{S}})$, for $i \in \{1, \ldots, n\} \setminus \{k\}$.

Note that the enacted system behavior $\mathcal{T}_{\mathcal{S}}$ is the synchronous product of the environment with the asynchronous product of the available behaviors. It is essentially the same form as any other enacted behavior, except for the presence of the index k in transitions. This makes explicit which behavior is the one that performs the action in the transition (while all other remain still).

8.3.3 Orchestrator and Composition

We can now introduce the notion of *orchestrator*, and define when it is a composition of the desired target service. The *orchestrator* is a component intended to activate, stop, and resume the available services (behaviors), and to instruct them to execute an action among those allowed in the current state. The orchestrator has *full observability* on the available behaviors and the environment, that is, it can keep track (at runtime) of their current states.

To formally define orchestrators, some technical notions are needed. A *trace* for an enacted behavior $\mathcal{T}_{\mathcal{B}}$ is a possibly infinite sequence of the form $s^0 \xrightarrow{a^1} s^1 \xrightarrow{a^2} \cdots$ such that (i) $s^0 = s_0$ and (ii) $s^j \xrightarrow{a^{j+1}} s^{j+1}$ in $\mathcal{T}_{\mathcal{B}}$, for all $j > 0$. A *history* is a finite prefix $h = s^0 \xrightarrow{a^1} \cdots \xrightarrow{a^\ell} s^\ell$ of a trace. We denote s^ℓ by $last(h)$, and ℓ by $length(h)$. The notions of trace and history extend immediately to enacted system behaviors: system traces have the form $s^0 \xrightarrow{a^1,k^1} s^1 \xrightarrow{a^2,k^2} \cdots$, and system histories have the form $s^0 \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^\ell,k^\ell} s^\ell$.

Let $\mathcal{S}$ be a system and $\mathcal{H}$ the set of its histories (i.e., histories of $\mathcal{T}_{\mathcal{S}}$). An *orchestrator* for $\mathcal{S}$ is a function $P : \mathcal{H} \times \mathcal{A} \mapsto \{1, \ldots, n, u\}$ that, given a history $h \in \mathcal{H}$ and an action $a \in \mathcal{A}$, selects a behavior, by returning its index, to delegate a to for execution. For technical convenience, a special value u ("undefined") can be returned, to make P a total function defined also on irrelevant histories or actions that no behavior can perform after a given history.

The problem we are interested in is the following: given a system $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E}\rangle$ and a deterministic *target* behavior $\mathcal{B}_t$ over $\mathcal{E}$, *synthesize an orchestrator P that realizes the target behavior $\mathcal{B}_t$ by suitably delegating each action requsted by*

$\mathcal{B}_t$ to one of the available behaviors $\mathcal{B}_i$in $\mathcal{S}$. A solution to such problem is called a *composition*.

Intuitively, the orchestrator realizes a target if for every trace of the enacted target and a requested action, the orchestrator returns the index of an available behavior able to perform the requested action. Observe that these orchestrators are somewhat akin to an advanced form of conditional plans and, in fact, the problem itself is related to planning, being both synthesis tasks. Here, though, plans do not select the next action, but *who shall execute the next action*.

One can formally define when an orchestrator realizes the target behavior, as in [18]. To this end, one first needs to define when an orchestrator P *realizes a trace* of the target $\mathcal{B}_t$. Then, since the target behavior is a deterministic transition system, and thus its behavior is completely characterized by the set of its traces, we can define that an *orchestrator P realizes the target behavior* $\mathcal{B}_t$ iff it realizes all of its traces.

8.3.4 Composition via Simulation

Let us next present our approach for synthesizing a composition, based on the notion of *simulation* [22]. Intuitively, a transition system S_1 "simulates" a system S_2 if S_1 is able to *match* all of S_2 moves. Due to the (devilish) nondeterminism of the available behaviors and the environment, we cannot use the off-the-shelf notion of simulation, but we need a variant, here called *ND-simulation*.

Let $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ be a system, $\mathcal{B}_t$ a target behavior over $\mathcal{E}$, and let $\mathcal{T}_\mathcal{S} = \langle S_\mathcal{S}, \mathcal{A}, \{1, \ldots, n\}, s_{\mathcal{S}0}, \delta_\mathcal{S} \rangle$ and $\mathcal{T}_t = \langle S_t, \mathcal{A}, s_{t0}, \delta_t \rangle$ the enacted system and enacted target behaviors corresponding to $\mathcal{S}$ and $\mathcal{B}_t$, respectively.

An *ND-simulation relation* of $\mathcal{T}_t$ by $\mathcal{T}_\mathcal{S}$ is a relation $R \subseteq S_t \times S_\mathcal{S}$, such that $\langle s_t, s_\mathcal{S} \rangle \in R$ implies:

1. $env(s_t) = env(s_\mathcal{S})$;
2. for all $a \in \mathcal{A}$, there exists a $k \in \{1, \ldots, n\}$ such that for all transitions $s_t \xrightarrow{a} s'_t$ in $\mathcal{T}_t$:
   - there exists a transition $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ with $env(s'_\mathcal{S}) = env(s'_t)$; and
   - for all transitions $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ with $env(s'_\mathcal{S}) = env(s'_t)$, we have $\langle s'_t, s'_\mathcal{S} \rangle \in R$.

In words, if a pair is in the ND-simulation, then (i) the component states share the same environment and (ii) for any possible move of the target from its state in the pair, there exists a behavior $\mathcal{B}_k$ able to match the move, while guaranteeing preservation of the ND-simulation.

We say that a state $s_t \in S_t$ *is ND-simulated by* a state $s_\mathcal{S} \in S_\mathcal{S}$ (or $s_\mathcal{S}$ *ND-simulates* s_t), denoted $s_t \preceq s_\mathcal{S}$, iff there exists an ND-simulation R of $\mathcal{T}_t$ by $\mathcal{T}_\mathcal{S}$ such that $\langle s_t, s_\mathcal{S} \rangle \in R$. Observe that this is a coinductive definition, thus the relation $\preceq$ is itself an ND-simulation, and in fact the *largest ND-simulation relation* w.r.t. set containment. Such a relation can be computed by the following *NDS* algorithm.

Roughly speaking, the algorithm works by iteratively removing those tuples for which the conditions of the ND-simulation definition do not apply.

Algorithm 1 $NDS(\mathcal{T}_t, \mathcal{T}_\mathcal{S})$ – Largest ND-Simulation

```
$\mathcal{R} := S_t \times S_\mathcal{S} \setminus \{\langle s_t, s_\mathcal{S}\rangle \mid env(s_t) \neq env(s_\mathcal{S})\}$
repeat
    $\mathcal{R} := (\mathcal{R} \setminus \mathcal{C})$, where $\mathcal{C}$ is the set of $\langle s_t, s_\mathcal{S}\rangle \in \mathcal{R}$ such that there exists $a \in \mathcal{A}$ for which for each
    $k$ there is a transition $s_t \xrightarrow{a} s'_t$ in $\mathcal{T}_t$ such that either:
    (a) there is no transition $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ such that $env(s'_t) = env(s'_\mathcal{S})$; or
    (b) there exists a transition $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ such that $env(s'_t) = env(s'_\mathcal{S})$ but $\langle s'_t, s'_\mathcal{S}\rangle \notin \mathcal{R}$.
until $(\mathcal{C} = \emptyset)$
return $\mathcal{R}$
```

The next result shows that checking for the existence of a composition can be reduced to checking whether there exists an ND-simulation between the enacted target and the enacted system that includes their respective initial states.

Theorem 8.1 *Let $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E}\rangle$ be a system and $\mathcal{B}_t$ a target behavior over $\mathcal{E}$. Let $\mathcal{T}_t = \langle S_t, \mathcal{A}, s_{t0}, \delta_t\rangle$ and $\mathcal{T}_\mathcal{S} = \langle S_\mathcal{S}, \mathcal{A}, \{1, \ldots, n\}, s_{\mathcal{S}0}, \delta_\mathcal{S}\rangle$ be the enacted target behavior and the enacted system behavior for $\mathcal{B}_t$ and $\mathcal{S}$, respectively. An orchestrator P for a system $\mathcal{S}$ that is a composition of the target behavior $\mathcal{B}_t$ over $\mathcal{E}$ exists iff $s_{t0} \preceq s_{\mathcal{S}0}$.*

Theorem 8.1 provides us with a straightforward procedure to check the existence of a composition. Namely, (i) compute the largest ND-simulation relation of $\mathcal{T}_t$ by $\mathcal{T}_\mathcal{S}$ and (ii) check whether $\langle s_{t0}, s_{\mathcal{S}0}\rangle$ occurs in the relation.

From the computational point of view, the algorithm *NDS* above computes the largest ND-simulation relation $\preceq$ between $\mathcal{T}_t$ and $\mathcal{T}_\mathcal{S}$ in polynomial time in the size of $\mathcal{T}_t$ and $\mathcal{T}_\mathcal{S}$. Since in our case the number of states of $\mathcal{T}_\mathcal{S}$ is exponential in the number of available behaviors $\mathcal{B}_1, \ldots, \mathcal{B}_n$, we get that $\preceq$ can be computed in exponential time in the *number of available behaviors*.

Theorem 8.2 *The existence of compositions can be checked in polynomial time in the number of states of the available behaviors, of the environment, and of the target behavior, and in exponential time in the number of available behaviors.*

Since the composition problem is EXPTIME-hard [32], the obtained bound is tight.

With the ND-simulation at hand we can *synthesize* an orchestrator. In fact, there is a well-defined procedure that, given an ND-simulation, builds a finite-state program that returns, at each point, the set of available behaviors capable of performing a target-conformant action, and guarantee the preservation of the ND-simulation. We call such a program *orchestrator generator*. Let $\mathcal{S}$ be a system, $\mathcal{B}_t$ a target behavior over $\mathcal{E}$, and let $\mathcal{T}_\mathcal{S}$ and $\mathcal{T}_t$ be the enacted system behavior and the enacted target behavior corresponding, respectively, to $\mathcal{S}$ and $\mathcal{B}_t$. The *orchestrator generator* of $\mathcal{S}$ for $\mathcal{B}_t$ is a tuple $OG = \langle \Sigma, \mathcal{A}, \{1, \ldots, n\}, \partial, \omega\rangle$, where:

1. $\Sigma = \{\langle s_t, s_\mathcal{S}\rangle \in S_t \times S_\mathcal{S} \mid s_t \preceq s_\mathcal{S}\}$ is the set of states of *OG*, formed by those pairs of $\mathcal{T}_t$ and $\mathcal{T}_\mathcal{S}$ states that are in the largest ND-simulation relation; given a state $\sigma = \langle s_t, s_\mathcal{S}\rangle$ we denote s_t by $com_t(\sigma)$ and $s_\mathcal{S}$ by $com_\mathcal{S}(\sigma)$.
2. $\mathcal{A}$ is the finite set of shared actions.
3. $\{1, \ldots, n\}$ is the finite set of available behavior indexes.
4. $\partial \subseteq \Sigma \times \mathcal{A} \times \{1, \ldots, n\} \times \Sigma$ is the *transition relation*, where $\langle \sigma, a, k, \sigma'\rangle \in \partial$, or $\sigma \xrightarrow{a,k} \sigma'$ in *OG*, iff
   - $com_t(\sigma) \xrightarrow{a} com_t(\sigma')$ in $\mathcal{T}_t$;
   - $com_\mathcal{S}(\sigma) \xrightarrow{a,k} com_\mathcal{S}(\sigma')$ in $\mathcal{T}_\mathcal{S}$;
   - for all $com_\mathcal{S}(\sigma) \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$, $\langle com_t(\sigma'), s'_\mathcal{S}\rangle \in \Sigma$.
5. $\omega : \Sigma \times A \mapsto 2^{\{1,\ldots,n\}}$ is the *output function*, where $\omega(\sigma, a) = \{k \mid \exists\, \sigma'$ s.t. $\sigma \xrightarrow{a,k} \sigma'$ in $OG\}$.

Thus, *OG* is a finite state transducer that, given an action *a* (compliant with the target behavior, and according to the system state corresponding to the current *OG* state), outputs, through ω, the set of *all* available behaviors that can perform *a* next while preserving the ND-simulation $\preceq$. Observe that computing *OG* from the relation $\preceq$ is easy, as it involves checking *local* conditions only.

Coming back to our example, when the user asks for action *hot air on*, the *OG* outputs the index that represents the bathroom heating device, which is the only one that can perform the requested action. If many bathrooms are available, thanks to the guards and to function ω, the composition layer can instruct one bathroom or another to perform the action, depending on realtime conditions, such as availability of a particular bathroom or device.

If there exists a composition of $\mathcal{B}_t$ by $\mathcal{S}$, then $s_{t0} \preceq s_{\mathcal{S}0}$ and *OG* does include the state $\sigma_0 = \langle s_{t0}, s_{\mathcal{S}0}\rangle$. In such cases, we get actual orchestrators, called *generated orchestrators*, which are compositions of $\mathcal{B}_t$ by $\mathcal{S}$, by picking up, at each step, one available behavior among those returned by ω. More precisely, we proceed as follows. A <u>*trace for OG*</u> starting from σ^0 is a finite or infinite sequence $\sigma^0 \xrightarrow{a^1,k^1} \sigma^1 \xrightarrow{a^2,k^2} \cdots$, such that $\sigma_j \xrightarrow{a^{j+1},k^{j+1}} \sigma_{j+1}$ in *OG*, for all j. A <u>*history for OG*</u> starting from state σ^0 is a prefix of a trace starting from σ^0. By using histories, one can introduce <u>*OG -orchestrators*</u>, which are functions $CGP_{\text{CHOOSE}} : \mathcal{H}_{OG} \times \mathcal{A} \mapsto \{1, \ldots, n, u\}$, where $\mathcal{H}_{OG}$ is the set of *OG* histories starting from any state in Σ, and defined as follows: $CGP_{\text{CHOOSE}}(h_{OG}, a) = \text{CHOOSE}(\omega(last(h_{OG}), a))$, for all $h_{OG} \in \mathcal{H}_{OG}$, where CHOOSE stands for a choice function that chooses one element among those returned by $\omega(last(h_{OG}), a))$. Assuming that *OG* (of $\mathcal{S}$ for $\mathcal{B}_t$) includes $\sigma_0 = \langle s_{t0}, s_{\mathcal{S}0}\rangle$, for any *OG* history $h_{OG} = \sigma^0 \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^\ell,k^\ell} \sigma^\ell$ starting from $\sigma^0 = \sigma_0$, we can obtain the corresponding system history $proj_\mathcal{S}(h_{OG})$, called the <u>*projected system history*</u>, as follows:

$proj_{\mathcal{S}}(h_{OG}) = com_{\mathcal{S}}(\sigma^0) \xrightarrow{a^1,k^1} \cdots \xrightarrow{a^\ell,k^\ell} com_{\mathcal{S}}(\sigma^\ell)$, i.e., we take the "system" component of each OG state σ^i in the history.

Moreover, from a OG-orchestrator CGP_{CHOOSE}, we obtain the corresponding <u>*generated orchestrator*</u> as the function $P_{\text{CHOOSE}} : \mathcal{H} \times \mathcal{A} \mapsto \{1, \ldots, n, u\}$, where $\mathcal{H}$ is the set of system histories starting from $s_{\mathcal{S}0}$, defined as follows. For each system history h and action a: (i) if $h = proj_{\mathcal{S}}(h_{OG})$ for some OG history h_{OG}, then $P_{\text{CHOOSE}}(h, a) = CGP_{\text{CHOOSE}}(h_{OG}, a)$; else (ii) $P_{\text{CHOOSE}}(h, a) = u$.

Through generated orchestrators, we can relate OGs to compositions and show that one gets *all* orchestrators that are compositions by considering all choice functions for CHOOSE. Notably, while each specific composition may be an infinite state program, the orchestrator generator OG, which includes all of them, is always finite.

We have the following central result, which states soundness and completeness of the orchestrator generation defined above.

Theorem 8.3 *If OG includes the state $\sigma_0 = \langle s_{t0}, s_{\mathcal{S}0}\rangle$, then every orchestrator generated by OG is a composition of the target behavior $\mathcal{B}_t$ by system $\mathcal{S}$. Moreover, every orchestrator that is a composition of the target behavior $\mathcal{B}_t$ by system $\mathcal{S}$ can be generated by OG.*

8.4 A Practical Application in Smart Homes

As previously stated, a concrete case of application of automated service composition with the Roman Model has been performed in the SM4ALL EU research project, recently and successfully concluded.[6]

8.4.1 Software Architecture, Service and Data Models

The goal of SM4ALL is to seamlessly integrate devices, in order to simplify the access to the services that they expose, and dynamically compose such services in order to offer the end users more complex functionalities and a richer experience with the domotic environment. In SM4ALL, all the devices make their functionalities available as SOAP-based Web services, according to a rich *service model*[7] consisting

[6] SM4All—Smart hoMes for All, is an FP7 project running from 1 September 2008 to 31 August 2011. Cf. the WWW site http://www.sm4all-project.eu/ and news on major international televisions: Globo TV—http://video.globo.com/Videos/Player/Noticias/0,,GIM1751401-7823-CASA+INTELIGENTE+E+MOVIDA+A+PENSAMENTO+NA+ITALIA,00.html, Channel 1 Russia—http://www.1tv.ru/news/other/191509, Italian Rai3—http://www.youtube.com/watch?v=a9F72_E4mT0 and http://rai.it/dl/tg3/rubriche/PublishingBlock-79554b45-1e4c-41a8-a474-ad3e22ab750f.html#, Ability Channel—http://www.abilitychannel.tv/video/casa-domotica-sm4all/ .

[7] Cf. http://www.dis.uniroma1.it/~cdc/sm4all/proposals/servicemodel/latest.

not only of the service interface specification, but also, e.g., of its conversational description and of the related graphical widgets (i.e., icons) to be presented in the user layer. *Proxies* are indeed the software components offering such services by “wrapping” and abstracting the real devices offering the functionalities. Services are not necessarily offered by hardware devices, but could be also realized through a human intervention; in this case, the proxy exposes a SOAP-based service to the platform, whereas it interacts with the service provider (i.e., the human) by means of a dedicated GUI, when executing the requested operations.

During their run time, services continuously change their status, both in terms of values of sensed/actuating variables (e.g., a service wrapping a temperature sensor reports the current detected temperature, a service wrapping windows blinds report whether the blinds are open, closed, half-way, etc.) and in terms of their conversational state. The definition of the sensed/actuating variables, representing the “state” of the domotic environment, is performed in accordance with the *data model*.[8]

The SM4ALL architecture, described in details in [20], consists of a *Pervasive Controller* and a *Discovery Framework*, which are in charge, when a new device joins the system, to dynamically load and deploy the appropriate service, and to register all the relevant information into the *Service Semantic Repository*. All of the status information, both in terms of (i) service conversational states and (ii) values of the environmental variables, are kept available in the *Context Awareness Manager*, through a publish&subscribe mechanism. On the basis of the service descriptions, *Composition Engines* are in charge of providing complex services by suitably composing the available ones. In SM4ALL, three different types of approaches are provided, each providing different functionalities and therefore complementing one another, in order to provide a rich and novel environment to the users:

- *Off-line synthesis* (provided through the Off-line Synthesis Engine). In the off-line mode, at design/deployment time of the house, a desiderata (i.e., not really existing) target service is defined, as a kind of complex routine, and the synthesis engine synthesizes a suitable orchestration of the available services realizing the target one. Such an orchestration specification is used at execution-time (i.e., when the user chooses to invoke the composite/desiderata service) by the Orchestration Engine in order to coordinate the available services (i.e., to interact with the user on one hand and to schedule service invocations on the other hand). In this approach, the orchestration specification is synthesized off-line (i.e., not triggered by user requests, at run time) and executed on-line as if it were a real service of the home. The off-line mode is based on the Roman Model. The Off-line Synthesis Engine produces what in SM4ALL is referred to as a *routine*.
- *On-line planning* (provided through the On-line Planning Engine). The user, during its interaction with the home, may decide not to invoke a specific service (either available/real or composite), but rather to ask the home to realize a *goal*; in such a case, the engine, on the basis of specific planning techniques [25],

[8] Cf. http://www.dis.uniroma1.it/~cdc/sm4all/proposals/datamodel/latest.

synthesizes and executes available service invocations in order to reach such a goal.

- *Visual design of complex services* (provided through the Compound Service Workbench). A skilled user may want to define a *compound service*, by visually composing services offered by proxies, in a way similar to what currently happen in technologies like WS-BPEL. The compound service offers an aggregated operation, which is the result of the proper orchestration of operations offered by other services. Also in this case, the synthesis is performed off-line, but differently from the previous case, it is not supported by automatic techniques, but by a visual workbench. Both routines and compound services fall under the category of "composite services".

The *Orchestration Engine* interprets the specification of a composite service (either synthesized automatically, through the Off-line Synthesis Engine, or visually by the user, through the Compound Service Workbench) and consequently orchestrates the set of component services. In the case of the On-line Planning Engine, due to the need of continuously planning and monitoring services during plan executions, the Orchestration Engine is bypassed and services are directly invoked by the planner itself.

Users are able to interact with the home and the platform through different kinds of user interfaces, e.g., a home control station accessible through a touchscreen in the living room. In particular, Brain Computer Interfaces (BCIs, [28]) allow also people with disabilities to interact with the system. Of course, users can still control the home equipment as if there were not the SM4ALL platform, e.g., a user is obviously allowed to switch the living room light on directly from the manual switcher on the wall, without using any BCI and/or touchscreen; in such a case, the platform, through the specific proxy wrapping the light/switcher as a service, is notified of the specific variable value change. De facto, the event is equivalent, due to the engineering of the platform, to the one of clicking a specific button on the touchscreen and/or selecting the icon on the BCI. Users are able, trough the interfaces, to invoke actions offered by services (either simple of composite) and to achieve goals, in order to reach specific situations that they would like to be realized in the home. Moreover, through the interfaces, they receive the feedback about state changes in the home, as well as requests for further inputs (in case additional parameters are needed for some actions to be executed), notifications about action/service completions, etc.

Going into implementation details of the Off-line Synthesis and Orchestration Engines, they have been realized as Java modules, realizing the techniques presented in Sect. 8.3. In particular, the Off-line Synthesis Engine is built around TLV (Temporal Logic Verifier),[9] an environment for verification of finite state systems; we defined a set of modules that make TLV compute the orchestration generator. Starting from XML descriptions of services (according to the service model), target service and variables, we had to devise a suitable translation into the TLV input language. After the orchestrator generator (see Sect. 8.3) has been computed, it is converted into our

[9] http://www.wisdom.weizmann.ac.il/~verify/tlv/

XML orchestration language (we named CBL—Composition Behavioral Language) which is interpreted by the Orchestration Engine, thus really executing at runtime the automatically synthesized composition. Further technical details can be found in [24].

As discussed in Sect. 8.3, the service model focuses on the behavior of services, in terms of conversational states that they traverse during the execution of the exposed actions, as well as on the way they (i) affect the environment and (ii) are inhibited (allowed) in the execution by the environment (respectively, by the expression of post-conditions and pre-conditions on top of the variables). The smart home environment is populated by many deployed *service instance*s, which are actual occurrences of given *service type*s (also *service*s for sake of brevity). Indeed a developer can produce many instances showing the same behavior: e.g., many lamps of the same product series, installed in different rooms, are different instances of the same service type. Therefore, every service instance can be identified by one or more *properties*, which are deployment characteristics (such as the location in the house, the power consumption, etc.).

The *data model* is an extensible framework of variable types. They concern the specific environmental information used by reasoning engines only. I.e., free parameters such as, e.g., `name` in an operation `cheers(name: string): string` may not adhere to the data model. Nevertheless, in case the developer wants (i) to describe the effects on the environment once a service action is invoked (post-condition), or (ii) to express the conditions that must hold in the context for an action to take place (pre-condition), she has to write statements formulated on top of variables whose type is coherent with the data model.

This is due to the fact that the platform should be able to cope with a predefined uniform set of common data types, so that the interaction with the environment is clear, despite of the service developer. We call *variable types* (or simply *types*) the types, and *variables* are the entities whose type is a *variable type*. The data model is an XML standard, i.e., it is based on XML Schemata to define value spaces. Each service developer can define her own types, provided that (i) they are described in XML Schema documents identified by a unique *namespace*, and (i) they extend, directly or indirectly, the SM4ALL *base type*s.[10] Indeed, types in the data model are derived by XML Schema native ones, and are designed to be extended by SM4ALL system service designers. The data model allows XML Schema simple types only as SM4ALL variable types, according to the XML Schema definition: complex types are not considered. Common variable types are enumerations on top of the `numeric` type. This allows the ordering over the possible values, as inherited from the basic integer type. In such cases, the insertion of a `documentation` tag for each enumerated value, provides also a human-readable form. The `documentation` node is intended to contain the information to (possibly) show the users. That is to say: if, e.g., a variable of type `temperatureLevel` reaches the value 3, the reasoning engines are informed of it, whereas the users are notified of a new "warm" status. Having enumerations over variables with finite sets of possible values makes feasible and effective the

[10] Base types are identified by the http://www.sm4all-project.eu/datamodel/base namespace.

reasoning tasks of the composition engines (as discussed in Sect. 8.3, the approach requires that the set of environmental states is finite).

8.4.2 Discussion and Lessons Learned

Applying automated composition in practice allowed to gain interesting lessons learned, about the performances and the acceptability of the approach by users. As stated in Sect. 8.3, computing an orchestrator generator is EXPTIME-complete, so an interesting question is which are the real dimensions of problems that can be practically solved by the approach. In the SM4ALL project, a testbed/showcase has been realized in a real domotic house located in Roma, Italy, equipped with about 20 sensors, some human-based services (e.g., a nurse assisting a disabled person) and some routines computed with the proposed approach. Table 8.1 reports the average times (over 20 runs) for computing such routines used in the testbed, by using the Off-line Synthesis Engine on a Intel Pentium 4 M, 512 Mb RAM, Ubuntu 10.04 32 bit. The available services amount to 18, whereas the column "services" reports how many services (over the 18 available) are effectively given as input of the problem. The reader should note the low features of the machine, as in a real smart home scenario, a platform like SM4ALL should run on a low-end hardware of type "set-top-box" (e.g., a multimedia player, an EEEBox, etc.) and not on an high-end server. Such times are appropriate, as the reader should remind that the routines are computed off-line, i.e., at design/deployment time of the smart home, and not during run-time, i.e., while living inhabitants exploit the platform.

In order to keep the number of variables and of services (effectively considered in the community given as input to the problem) as low as possible, a careful decomposition approach should be undertaken when defining service descriptions. The reader should note that if a variable should be considered in a composition, then also all possible services affecting such a variable should be considered as input to the problem. Indeed in our testbed, naively the *nurse* service was affecting

Table 8.1 Average times for synthesis

Target	Transitions	States	Variables	Services	Avrg. time (millis)
WakeUp	332	84	3	5	1240.9
CheckIngredients	2704	352	5	3	3340.8
SetAlarm	48	29	1	3	305.3
WakeUpLite	101	41	2	4	487.75
EscapeRoutine	597	93	3	4	1126
RestoreFromWakeUp	85	25	2	3	320.5
RelaxModeSetup	4437	357	4	5	7806.3
FootbalMatchSetupN	2263	215	4	5	3257.75
FootbalMatchSetupW	301	53	3	4	643.95

7 variables, with the result that all routines require, during the composition, to consider as input community all the 18 available services, finally making the computation not practically feasible (after 3 days of running, no composition has been computed yet). Conversely, considering *nurse4bedroom*, *nurse4kitchen* and *nurse4livingRoom* distinct services, each one affecting different variables, we were finally able to keep the number of input services and of variables low, obtaining the above results.

As far as user acceptability, assuming that a designer is willing to provide a target service as input for the composition revealed difficult in many cases, especially if the target is quite complex; on the other side, defining goals is widely accepted, even if in many cases a more fine-grained control over possible intermediate goals is desiderated. To this aim, we started investigating a novel model, which in some sense merges the conversational approach of the Roman model with the "goal-based" approach typical of automated composition based on planning; preliminary results can be found in [14, 15, 17].

Acknowledgments The authors would like to thank all the persons who contributed over the years to the Roman model: Daniela Berardi, Diego Calvanese, Maurizio Lenzerini, Richard Hull, Alessandro Iuliani, Damiano Pozzi, Fahima Cheikh, Valerio Colaianni, Sebastian Sardiña, Claudio Di Ciccio, Riccardo De Masellis, Paolo Felli, Ettore Iacomussi, Vincenzo Forte, Mario Caruso. We also would like to acknowledge the support of the projects MAIS and Brindisys (Italian), SemanticGov and TONES (EU FP6), SM4All, GreenerBuildings and ACSI (EU FP7).

References

1. Baligand, F., Rivierre, N., Ledoux, T.: A declarative approach for QoS-aware web service compositions. In: Proceedings of ICSOC (2007)
2. Beauche, S., Poizat, P.: Automated service composition with adaptive planning. In: Proceedings of ICSOC (2008)
3. Benatallah, B., Casati, F., Toumani, F.: Web service conversation modeling: a cornerstone for e-business automation. IEEE Internet Comput. **8**(1), 46–54 (2004)
4. Benatallah, B., Sheng, Q.Z., Dumas, M.: The Self-Serv environment for web services composition. IEEE Internet Comput. **7**(1), 40–48 (2003)
5. Berardi, D., Calvanese, D., De Giacomo, G., Hull, R., Mecella, M.: Automatic composition of transition-based semantic web services with messaging. In: Proceedings of VLDB (2005)
6. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic composition of e-Services that export their behavior. In: Proceedings of ICSOC (2003)
7. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Synthesis of underspecified composite e-Services based on automated reasoning. In: Proceedings of ICSOC (2004)
8. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic service composition based on behavioural descriptions. Int. J. Coop. Inf. Syst. **14**(4), 333–376 (2005)
9. Berardi, D., Calvanese, D., De Giacomo, G., Mecella, M.: Composition of services with nondeterministic observable behavior. In: Proceedings of ICSOC (2005)
10. Berardi, D., Cheikh, F., De Giacomo, G., Patrizi, F.: Automatic service composition via simulation. Int. J. Found. Comput. Sci. **19**(2), 429–451 (2008)
11. Bultan, T., Fu, X., Hull, R., Su, J.: Conversation specification: a new approach to design and analysis of e-Service composition. In: Proceedings of WWW (2003)

12. Cardoso, J., Sheth, A.: Introduction to semantic web services and web process composition. In: Proceedings of 1st International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004) (2004)
13. Cheikh, F., De Giacomo, G., Mecella, M.: Automatic web services composition in trustaware communities. In: Proceedings of 3rd ACM Workshop On Secure Web Services (SWS 2006) (2006)
14. De Giacomo, G., Di Ciccio, C., Felli, P., Hu, Y., Mecella, M.: Goal-based composition of stateful services for smart homes. In: Proceedings of CoopIS (2012)
15. De Giacomo, G., Felli, P., Patrizi, F., Sardiña, S.: Two-player game structures for generalized planning and agent composition. In: Proceedings of AAAI (2010)
16. De Giacomo, G., De Masellis, R., Patrizi, F.: Composition of partially observable services exporting their behaviour. In: Proceedings of ICAPS (2009)
17. De Giacomo, G., Patrizi, F., Sardiña, S.: Agent programming via planning programs. In: Proceedings of AAMAS (2010)
18. De Giacomo, G., Sardiña, S.: Automatic synthesis of new behaviors from a library of available behaviors. In: Proceedings of IJCAI (2007)
19. De Paoli, F., Lulli, G., Maurino, A.: Design of quality-based composite web services. In: Proceedings of ICSOC (2006)
20. Di Ciccio, C., Mecella, M., Caruso, M., Forte, V., Iacomussi, E., Rasch, K., Querzoni, L., Santucci, G., Tino, G.: The homes of tomorrow: service composition and advanced user interfaces. ICST Trans. Ambient Syst. **11**(10–12), e2 (2011)
21. Gerede, C., Hull, R., Ibarra, O.H., Su, J.: Automated composition of e-Services: Lookaheads. In: Proceedings of ICSOC (2004)
22. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: Proceedings of FOCS (1995)
23. Hull, R.: Web services composition: a story of models, automata, and logics. In: Proceedings of SCC (2005)
24. Iacomussi, E.: Service-based architectures for smart homes and the SM4All project. The component for the automatic synthesis of conversational services. Master thesis, Sapienza Università di Roma (2011). A copy can be obtained by writing an email to authors
25. Kaldeli, E., Lazovik, A., Aiello, M.: Extended goals for composing services. In: Proceedings of ICAPS (2009)
26. Klein, A., Ishikawa, F., Honiden, S.: Efficient QoS-aware service composition with a probabilistic service selection policy. In: Proceedings of ICSOC (2010)
27. Lenzerini, M.: Data integration: a theoretical perspective. In: Proceedings of PODS (2002)
28. McFarland, D.J., Wolpaw, J.R.: Brain-computer interfaces for communication and control. Commun. ACM **54**(5), 60–66 (2011)
29. McIlraith, S., Son, T.: Adapting golog for composition of semantic web services. In: Proceedings of KR (2002)
30. Medjahed, B., Bouguettaya, A., Elmagarmid, A.: Composing web services on the semantic web. Very Large Data Base J. **12**(4), 333–351 (2003)
31. Michalowski, M., Ambite, J., Thakkar, S., Tuchinda, R., Knoblock, C., Minton, S.: Retrieving and semantically integrating heterogeneous data from the web. IEEE Int. Syst. **19**(3), 72–79 (2004)
32. Muscholl, A., Walukiewicz, I.: A lower bound on web services composition. Logical Methods Comput. Sci. **4**(5), 1–14 (2008). doi:10.2168/LMCS-4, http://www.lmcs-online.org/ojs/viewarticle.php?id=359
33. Pistore, M., Marconi, A., Bertoli, P., Traverso, P.: Automated composition of web services by planning at the knowledge level. In: Proceedings of IJCAI (2005)
34. Sardiña, S., De Giacomo, G., Patrizi, F.: Behavior composition in the presence of failure. In: Proceedings of KR (2008)
35. Sardiña, S., Patrizi, F., De Giacomo, G.: Automatic synthesis of a global behavior from multiple distributed behaviors. In: Proceedings of AAAI (2007)

36. Schuller, D., Miede, A., Eckert, J., Lampe, U., Papageorgiou, A., Steinmetz, R.: Qos-based optimization of service compositions for complex workflows. In: Proceedings of ICSOC (2010)
37. Wang, H., Zhou, X., Zhou, X., Liu, W., Li, W., Bouguettaya, A.: Adaptive service composition based on reinforcement learning. In: Proceedings of ICSOC (2010)
38. Wu, D., Parsia, B., Sirin, E., Hendler, J., Nau, D.: Automating DAML-S web services composition using SHOP2. In: Proceedings of ISWC (2003)
39. Yang, J., Papazoglou, M.: Service components for managing the life-cycle of service compositions. Inf. Syst. **29**(2), 97–125 (2004)
40. Zhao, H., Doshi, P.: A hierarchical framework for composing nested web processes. In: Proceedings of ICSOC (2006)