Goal-Based Composition of Stateful Services for Smart Homes

Giuseppe De Giacomo[1], Claudio Di Ciccio[1], Paolo Felli[1],
Yuxiao Hu[2], and Massimo Mecella[1]

[1] Sapienza – Università di Roma, Italy
{degiacomo,cdc,felli,mecella}@dis.uniroma1.it
[2] Google Waterloo, Canada
yuxiao@google.com

Abstract. The emerging trend in process management and in service oriented applications is to enable the composition of new distributed processes on the basis of user requests, through (parts of) available (and often embedded in the environment) services to be composed and orchestrated in order to satisfy such requests. Here, we consider a user process as specified in terms of repeated goals that the user may choose to get fulfilled, organized in a kind of routine. Available services are suitably composed and orchestrated in order to realize such a process. In particular we focus on smart homes, in which available services are those ones offered by sensor and actuator devices deployed in the home, and the target user process is directly and continuously controlled by the inhabitants, through actual goal choices. We provide a solver that synthesizes the orchestrator for the requested process and we show its practical applicability in a real smart home use case.

Keywords: process/service composition, smart houses/buildings, planning techniques.

1 Introduction

The promise of Web services (WSs) and of Service Oriented Architectures (SOAs) in general, coupled with the technologies and methodologies of Business Process Management (BPM), is to enable the composition of new distributed processes/solutions: (parts of) available services can be composed and orchestrated in order to realize complex processes, offering advanced functionalities to users.

Many approaches (surveyed, e.g., in [3]) have been proposed in the last years in order to address the above problem from different viewpoints. Works based on Planning in AI, such as [16,24,5,26] consider only the input/output specification of available services, which is captured by atomic actions together with their pre- and post-conditions (a notable extension is [2]), and specify the overall semantics in terms of propositions/formulas (facts known to be true) and actions, affecting the proposition values. All these approaches consider *stateless*

R. Meersman et al. (Eds.): OTM 2012, Part I, LNCS 7565, pp. 194–211, 2012.

services, as the operations offered to clients do not depend on the past history, as services do not retain any information about past interactions. Also other works (e.g., [25,17,8,6]) consider available services as atomic actions, but, rather than on (planning-based) composition, they focus on modeling issues and automatic service discovery, by resorting to rich ontologies as a basic description mean. Many works (e.g., [15,22,1,18] consider how to perform composition by taking into account Quality-of-Service (QoS) of the composite and component services. Some works consider non classical techniques (e.g., [23] adopts learning approaches) for solving the composition problem.

There are also approaches (e.g., [12]) that consider *stateful* services, which impose constraints on the possible sequences of interactions (a.k.a., conversations) that a client can engage with the service. Stateful services raise additional challenges, as the process coordinating such services should be correct w.r.t. the possible conversations allowed by the services themselves. An interesting approach of this type is the one of [19], in which the specification is a set of atomic actions and propositions, like in planning, services are (finite-state) transition systems whose transitions correspond to action executions, which, in general, affect the truth values of propositions, and the client requests a (main) goal (i.e., a formula built from the above propositions) to be achieved, while requiring runtime failures to be properly handled by achieving a special exception handling goal. Another interesting approach is the one adopted in [4], often referred to as *Roman Model*, in which again 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.

In this paper, we consider a notable extension of the Roman Model, where *goal-based processes* are used, instead of target services, to specify what the user desires to achieve. Such processes can be thought of as *routines* built from virtual tasks expressed declaratively simply as *goals*, which allow users to specify the desired state of affair to bring about. Such goals are organized in a control flow structure, possibly involving loops that regulates their sequencing, as well as the decision points where the user can choose the next goal to request.

Wrt [19], the main novelty proposed in this paper is allowing clients to request new goals, once the current one is achieved (by a plan). In general, such requests can be arranged as routines represented as finite-state transition systems whose transitions correspond to goal requests. These routines typically involve loops, thus ruling out naïve approaches based on (classical, conditional or conformant) planning. Indeed, not all plans that achieve a requested goal are successful: some might lead the system to states preventing future client requests fulfillment. Such *bad* plans could be recognized by taking into account *all* goals the client can request in the future, which, in the presence of loops, span over an infinite horizon (though finite-state).

The approach proposed here is strongly motivated by challenging applications in the domain of smart houses and buildings, i.e., buildings pervasively equipped with sensors and actuators making their functionalities available according to the service-oriented paradigm. In order to be dynamically configurable and composable, embedded services need to expose semantically rich service descriptions, comprising *(i)* interface specifications and *(ii)* specifications of the externally visible behaviors. Moreover, human actors in the environment can be abstracted as services, and actually "wrapped" by a semantic description (e.g., a nurse offering medical services). This allows them to be involved in orchestrations and to collaborate with devices, to reach certain goals. See, e.g., [14,11,7].

We envision a user that can express processes she would like to have realized in the house, in the form of routines consisting of goals (e.g., states of the house she would like to have realized); an engine automatically synthesizes the right orchestration of services able to satisfy the goals. Users can interact with the house through different kinds of interfaces, either centralized (e.g., in a home control station) or distributed, and embedded in specific interface devices. Brain Computer Interfaces (BCIs) allow also people with disabilities to interact with the system. Using such interfaces, users issue specific goals to the system, which is, in turn, expected to react and satisfy the request.

In this paper, we detail this approach. We provide a framework for composition of goal-oriented processes form available (non-atomic) services (Section 2). We present a case study where the framework is applied in a real smart home setting (Section 3). We provide a effective solver (Section 4), which synthesize an orchestrator that realizes the target goal-oriented processes, by detailing subprocesses that fulfil the various goals at the various point in time. Our solver is sound and complete, and far more practical than other solutions proposed in literature, also because it easily allows for exploiting heuristic in the search for the solution. We show the effectiveness of our solver with some experiments in our use case (Section 5). We conclude the paper with a brief discussion on further work (Section 6).

2 Framework

We assume that the user acts on an environment that is formalized as a possibly nondeterministic *dynamic domain* $\mathcal{D}$, which provides a symbolic abstraction of the world that the user acts in. Formally, a *dynamic domain* is a tuple $\mathcal{D} = \langle P, A, D_0, \rho \rangle$, where:

- $P = \{p_1, \ldots, p_n\}$ is a finite set of *domain propositions*. $D \in 2^P$ is a *state*;
- $A = \{a_1, \ldots, a_r\}$ is the finite set of *domain actions*;
- $D_0 \in 2^P$ is the *initial state*;
- $\rho \subseteq 2^P \times A \times 2^P$ is the *transition relation*. We freely interchange notations $\langle D, a, D' \rangle \in \rho$ and $D \xrightarrow{a} D'$ in $\mathcal{D}$.

Intuitively, a dynamic domain models an environment whose states are described by the set P of boolean propositions, holding all relevant information about the current situation. For instance, the state of a room can be defined by the light being on or off and the door being open or closed, using two propositions *light_on* and *door_open*. By convention, we say that if one of such propositions is in the current state of $\mathcal{D}$, then it evaluates to $\top$ (true), otherwise it is $\bot$ (false). Hence, a propositional formula φ over P *holds* in a domain state $D \in 2^P$ ($D \models \varphi$) if φ evaluates to $\top$ when all of its propositions occurring in D are replaced by $\top$. However, such domain can not be manipulated directly, i.e., domain actions can not be accessed directly by the user: they are provided through *available services.* The idea is that, at each moment, a service offers a set of possible actions, and the user can interact with the domain $\mathcal{D}$ *only* by means of available services.

Given a dynamic domain $\mathcal{D}$, a *service* over $\mathcal{D}$ is a tuple $\mathcal{B} = \langle B, O, b_0, \varrho \rangle$, where: *(i)* B is the finite set of service states; *(ii)* O is the finite set of service actions over the domain, i.e., $O \cap A \neq \emptyset$; *(iii)* $b_0 \in B$ is the service initial state; *(iv)* $\varrho \subseteq B \times O \times B$ is the service transition relation. We will interchange notations $\langle b, a, b' \rangle \in \varrho$ and $b \xrightarrow{a} b'$ in $\mathcal{B}$.

As a service is instructed to perform an action over $\mathcal{D}$, both the service and the domain evolve *synchronously* (and possibly *nondeterministically*) according to their respective transition relations. So, for a domain action to be carried out, it needs to be both compatible with the domain and (currently) available in some service. However, services can also feature *local* actions, i.e., actions whose execution does not affect the domain evolution. For instance, a service representing a physical device might require to be switched on to use all its functionalities, a fact that is not captured by $\mathcal{D}$ alone. To define formally this idea, we introduce the notiton of executability for actions of a service $\mathcal{B} = \langle B, O, b_0, \varrho \rangle$: given $\mathcal{B}$ in its own service state b and a domain $\mathcal{D}$ in domain state D, action $a \in O$ is said to be *executable* by $\mathcal{B}$ in b iff *(i)* it is available in b, i.e. $b \xrightarrow{a} b'$ in $\mathcal{B}$ for some state $b' \in B$ and *(ii)* it is either a local action ($a \notin O \cap A$) or it is allowed in D, i.e., there exists a domain state D' such that $D \xrightarrow{a} D'$. Notice that services are loosely-coupled with the domain they are interacting with: new services can be easily added to the systems and modifications to the description of the underlying domain do not affect them.

Example 1. Consider a dynamic domain $\mathcal{D} = \langle P, A, D_0, \rho \rangle$ describing (among othet components) a simple door as in Figure 2a. A domain proposition *doorIsOpen* $\in P$ is used to keep its state, and the door can be either closed or opened executing domain actions $\{\mathsf{doClose}, \mathsf{doOpen}\} \subseteq A$. However, the door can only be managed through a service *doorSrv* $= \langle \{\mathsf{open}, \mathsf{closed}\}, \{\mathsf{doOpen}, \mathsf{doClose}\}, \mathsf{open}, \varrho \rangle$ where ϱ is such that $\mathsf{open} \xrightarrow{\mathsf{doClose}} \mathsf{closed}$ and $\mathsf{closed} \xrightarrow{\mathsf{doOpen}} \mathsf{open}$. Assume that *doorSrv* is in its state open, and the current domain state D to be such that *doorIsOpen* $\in D$ (i.e., it evaluates to true in D). As soon as action doClose is executed, both the doorSrv service evolves changing its state to closed and, *synchronously*, the domain evolves to a state D' such that *doorIsOpen* $\notin D'$ (i.e., it evaluates to false in D').

Given a dynamic domain $\mathcal{D}$ and a fixed set of available services over it, we define a *dynamic system* to be the resulting global system, seen as a whole: it is an abstract structure used to capture the interaction of available services with the environment. Formally, given a dynamic domain $\mathcal{D}$ and a set of available services $\mathcal{B}_1, \ldots, \mathcal{B}_n$, with $\mathcal{B}_i = \langle B_i, O_i, b_{i0}, \varrho_i \rangle$, the corresponding *dynamic system* is the tuple $\mathcal{S} = \langle S, \Gamma, s_0, \vartheta \rangle$, where:

- $S = (B_1 \times \cdots \times B_n) \times 2^P$ is the set of system states;
- $\mathit{\Gamma} = A \cup \bigcup_{i=1}^{n} O_i$ is the set of system actions;
- $s_0 = \langle\langle b_{10}, \ldots, b_{n0}\rangle, D_0\rangle \in S$ is the system initial state;
- $\vartheta \subseteq S \times (\mathit{\Gamma} \times \{1, \ldots, n\}) \times S$ is the system transition relation such that $\langle\langle b_1, \ldots b_n\rangle, D\rangle \xrightarrow{a,i} \langle\langle b'_1, \ldots b'_n\rangle, D'\rangle$ is in ϑ iff:
  (i) $\langle b_i, a, b'_i\rangle \in \varrho_i$;
  (ii) for each $j \in \{1, \ldots, n\}$, if $j \neq i$ then $b'_j = b_j$.
  (iii) if $a \in A$ then $\langle D, a, D'\rangle \in \rho$, otherwise $D' = D$;

(i)-*(ii)* require that only one service $\mathcal{B}_i$ moves from its own state b_i to b'_i performing action a, and *(iii)* requires that, if the action performed is not a local action, the domain evolves accordingly. Indeed, the set of system operations $\mathit{\Gamma}$ includes operations local to services, i.e. whose execution, according to ϑ, does not affect the domain evolution. We stress the fact that a dynamic system does not correspond to any actual structure: it is a convenient representation of the interaction between the available services and the domain. Indeed, a dynamic system captures the joint execution of a dynamic domain and a set of services where, at each step, only one service moves, and possibly affects, through operation execution, the state of the underlying domain. The evolutions of a system $\mathcal{S}$ are captured by its *histories*, herehence $\mathcal{S}$*-histories*. One such history is a finite sequence of the form $\tau = s^0 \xrightarrow{a^1, j^1} s^1 \cdots s^{\ell-1} \xrightarrow{a^\ell, j^\ell} s^\ell$ of length $|\tau| \doteq \ell + 1$ such that *(i)* $s^i \in S$ for $i \in \{0, \ldots, \ell\}$; *(ii)* $s^0 = s_0$; *(iii)* $s^i \xrightarrow{a^{i+1}, j^{i+1}} s^{i+1}$ in $\mathcal{S}$, for each $i \in \{0, \ldots, \ell - 1\}$. We denote with $\tau|_k$ its k-length (finite) prefix, and with $\mathcal{H}$ the set of all possible $\mathcal{S}$-histories. Given a dynamic system $\mathcal{S}$, a *general plan* is a (possibly partial) function $\pi : \mathcal{H} \longrightarrow \mathit{\Gamma} \times \{1, \ldots, n\}$ that outputs, given an $\mathcal{S}$-history, a pair representing the action to be executed and the index of the service which has to execute it. An *execution* of a general plan π from a state $s \in S$ is a possibly infinite sequence $\tau = s^0 \xrightarrow{a^1, j^1} s^1 \xrightarrow{a^2, j^2} \cdots$ such that *(i)* $s^0 = s$; *(ii)* $\tau|_k$ is an $\mathcal{S}$-history, for all $0 < k \leq |\tau|$; and *(iii)* $\langle a^k, j^k\rangle = \pi(\tau|_k)$, for all $0 < k < |\tau|$. When all possible executions of a general plan are finite, the plan is a *conditional plan*. The set of all conditional plans over $\mathcal{S}$ is referred to as $\mathit{\Pi}$. Note that, being finite, executions of conditional plans are $\mathcal{S}$-histories. A finite execution τ such that $\pi(\tau)$ is undefined is a *complete execution*, which means, informally, that the execution cannot be extended further. In the following, we shall consider only conditional plans.

We say that an execution $\tau = s^0 \xrightarrow{a^1, j^1} s^1 \cdots s^{\ell-1} \xrightarrow{a^\ell, j^\ell} s^\ell$ of a conditional plan π, with $s^i = \langle\langle b^i_1, \ldots b^i_n\rangle, D^i\rangle$:

- *achieves* a goal ϕ iff $D^\ell \models \phi$
- *maintains* a goal ψ iff $D^i \models \psi$ for every $i \in \{0, \ldots, \ell - 1\}$

Such notions can be extended to conditional plans: a conditional plan π *achieves* ϕ from state s if all of its complete executions from s do so; and π *maintains* ψ from s if all of its (complete or not) executions from s do.

Finally, we can formally define the notion of *(goal-based) target process* for a dynamic domain $\mathcal{D}$ as a tuple $\mathcal{T} = \langle T, \mathcal{G}, t_0, \delta\rangle$, where:

- $T = \{t_0, \ldots, t_q\}$ is the finite set of *process states*;
- $\mathcal{G}$ is a finite set of goals of the form *achieve ϕ while maintaining ψ*, denoted by pairs $g = \langle \psi, \phi \rangle$, where ψ and ϕ are propositional formulae over P;
- $t_0 \in T$ is the *process initial state*;
- $\delta \subseteq T \times \mathcal{G} \times T$ is the *transition relation.* We will also write $t \xrightarrow{g} t'$ in $\mathcal{P}$.

A target process $\mathcal{T}$ is a transition system whose states represent *choice points*, and whose transitions specify pairs of *maintenance* and *achievement* goals that the user can request at each step. Hence, $\mathcal{T}$ allows to combine achievement and maintenance goals so that they can be requested (and hence fulfilled) according to a specific temporal arrangement, which is specified by the relation δ of $\mathcal{T}$. Intuitively, a target process $\mathcal{T}$ is *realized* when a conditional plan π is available for the goal couple $g = \langle \phi, \psi \rangle$ chosen from initial state t_0 and, upon plan's completion, a new conditional plan π' is available for the new selected goal, and so on. In other words, all potential target requests respecting $\mathcal{T}$'s structure (possibly infinite) have to be fulfilled by a conditional plan, which is meant to be executed starting from the state that previous plan execution left the dynamic system $\mathcal{S}$ in (initially from s_0). Since the sequences of goals actually chosen by the user can not be forseen, a realization has to take into account all possible ones: at any point in time, all possible choices available in the target process must be guaranteed by the system, i.e., every legal request needs to be satisfied. We are going to give a formal definition of this intuition [9] in the remainder of this section.

Let $\mathcal{S}$ be a dynamic system and $\mathcal{T}$ a target process. A *PLAN-simulation relation*, is a relation $R \subseteq T \times S$ such that $\langle t, s \rangle \in R$ implies that for each transition $t \xrightarrow{\langle \psi, \phi \rangle} t'$ in $\mathcal{T}$, there exists a conditional plan π such that: *(i)* π *achieves* ϕ and *maintains* ψ from state s and *(ii)* for all π's possible complete executions from s of the form $s \xrightarrow{\pi(\tau|_1)} \cdots \xrightarrow{\pi(\tau|_\ell)} s^\ell$, it is the case that $\langle t', s^\ell \rangle \in R$. A plan π *preserves* R from $\langle t, s \rangle$ for a given transition $t \xrightarrow{\langle \psi, \phi \rangle} t'$ in $\mathcal{T}$ if requirement *(ii)* above holds. Also, we say that a target process state $t \in T$ is *PLAN-simulated* by a system state $s \in S$, denoted $t \preceq_{PLAN} s$, if there exists a *PLAN*-simulation relation R such that $\langle t, s \rangle \in R$. Moreover, we say that a target process $\mathcal{T}$ is *realizable* in a dynamic system $\mathcal{S}$ if $t_0 \preceq_{PLAN} s_0$. When the target process is realizable, one can compute *once for all* (*offline*) a function $\Omega : S \times \delta \longrightarrow \Pi$ that, if at any point in time the dynamic system reaches state s and the process requests transition $t \xrightarrow{\langle \psi, \phi \rangle} t'$ of $\mathcal{T}$, outputs a conditional plan π that its execution starting from s *(i)* achieves ϕ while maintaining ψ and *(ii)* preserves $\preceq_{PLAN}$, i.e., it guarantees that, for all possible states the system can reach upon π's execution, all target transitions outgoing from t' (according to δ) can still be realized by a conditional plan (possibly returned by the function itself). Such function Ω is referred to as *process realization.*

We can now formally state the problem of concern: *Given a dynamic domain $\mathcal{D}$, available services $\mathcal{B}_1, \ldots, \mathcal{B}_n$, and a target process $\mathcal{T}$, build, if it exists, a realization of $\mathcal{T}$ in the dynamic system $\mathcal{S}$ corresponding to $\mathcal{D}$ and $\mathcal{B}_1, \ldots, \mathcal{B}_n$.* In previous work [9,10], a solution to a simplified variant of our problem has been proposed[1]. Here, as discussed above, we explicitly distinguish between dynamic domain and available services, thus obtaining a different, more sophisticated problem. Nonetheless, the techniques presented there still apply, as we can reduce our problem to that case. This allows us to claim this result:

Theorem 1 ([9]). *Building a realization of a target process $\mathcal{T}$ in a dynamic system $\mathcal{S}$ is an EXPTIME-complete problem.*

3 Case Study

Here we present a case study, freely inspired by a real storyboard of a live demo held in a smart home located in Rome, whose houseplant is depicted in Figure 1.

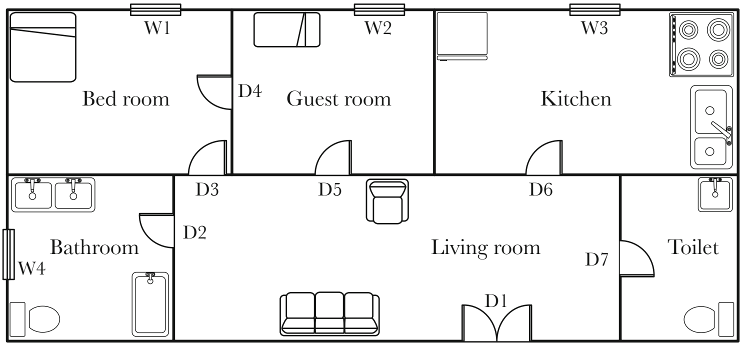


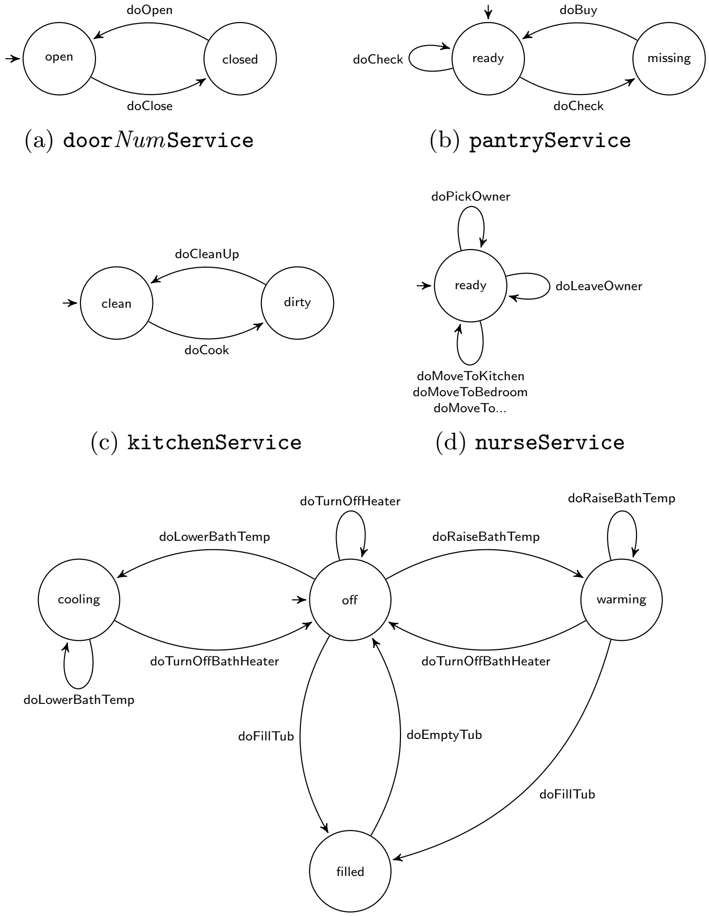
Fig. 1. The smart home plant

The home is equipped with many devices and a central reasoning system, whose domotic core is based on the framework and solver described in this paper, in order to orchestrate all the offered services. Imagine here lives Niels, a man affected by Amyotrophic Lateral Sclerosis (ALS). He is unable to walk, thus he needs a wheelchair to move around the house. The other human actors are Dan, a guest sleeping in the living room, and Wilma, the nurse. At the beginning of the story, Niels is sleeping in his automated bed.

The services the system can manage are the `bedService`, i.e., an automated bed, which can be either `down` or `up`; the `door`*Num*`Service`, i.e., the doors, for *Num* $\in \{1, 7\}$ (Figure 2a); the `alarmService`, i.e., an alarm, that can be either `set` or not; the `light`*Room*`Service`, i.e., light bulbs and lamps, for each *Room* in Figure 1; the `kitchenService`, i.e., a cooking service with preset dishes (Figure 2c); the `pantryService`, i.e., an automated pantry, able to check whether ingredients are in or not and buy them, if missing (see Figure 2b); the `bathroomService`,

[1] In particular, in [9,10] services are modelled directly in terms of restrictions on the domain.

i.e., a bathroom management system, able to warm the temperature inside and fill or empty the tub (Figure 2e); and finally the tvService, i.e., a TV, either on or off.



(a) door*Num*Service

(b) pantryService

(c) kitchenService

(d) nurseService

(e) bathroomService

Fig. 2. Case study services

Finally, we consider a very particular service, that we call nurseService: it is Wilma, the nurse, who is in charge of moving Niels around the house. Despite the fact that an analogous service could be provided by some mechanical device, we refer to an human to illustrate how actors can be abstracted as services as well, wrapped by a semantic description. All services are depicted in Figure 2, except for light*Room*Service, bedService, alarmService and tvService that have very simple on/off behaviors. As described in Section 2, a dynamic domain state is a subset of 2^P, where $P = \{p_1, \ldots, p_n\}$ is a finite set of boolean domain propositions. In order to express that, e.g., the bathroom temperature is mild, we could make use of a grounded propositional letter such as *bathroomTemperatureIsMild*. Nevertheless, we would have a grounded proposition for each value that the sensed temperature may assume (*bathroomTemperatureIsHot*, etc.) with the implicit constraint that only one of them can be evaluated to $\top$ at a time (and all the others to $\bot$). Thus, for sake of simplicity, here we make use of statements of the form "*var* = *val*" (e.g., "*varBathroomTemperature* = *warm*"). We call *var* the domain variable; *val* can be equal to any expected value which *var* can assume. Using such abbreviations we can phrase concepts like "a domain variable *var* is *set to* the *val* value" to easily refer to a transition in the dynamic domain moving from the current state to a following one where the proposition *var* = *val* holds. For the sake of readability, actions are identified by the do-prefix (e.g., doRing).

Now we comment the case study. All services affect, through their actions, the related domain variables representing the state of the context. As an example, consider Figure 2e. Action doRaiseBathTemp causes the `bathroomService` to reach warming state, and affects the domain setting *varBathroomTemperature* either to *(i) mild* if it was equal to *cold*, or *(ii) warm*, if previously *mild*. However, we can imagine also *indirect* effects: e.g., the `door4Service` and `door5Service`'s doOpen actions trivially turn the *varDoor4* and *varDoor5* domain variables from *closed* to *open* and, at the same time, change the *varGuestDisturbed* domain variable from *false* to *true*, since, as depicted in Figure 1, they leed to the guest room, which we supposed Dan, the guest, to sleep in. The dynamic domain constrains the execution of service actions, allowing *executable* transitions only to take place (as explained in Section 2). For instance, consider the doPick and doLeave actions in `nurseService`: they represent Wilma taking and releasing Niels' wheelchair. Even if they are always available according to the service's description (Figure 2d), they are allowed by the domain iff *varPositionOwner* and *varPositionNurse* are equal (i.e., iff $\bigvee_{r \in Rooms}(varPositionOwner = r \wedge varPositionNurse = r)$ for $Rooms = \{livingRoom, bedRoom, bathRoom, guestRoom, toilet, kitchen\}$). Further on, it is stated that you can activate the doPick transition only if *varOwnerPicked* = *false* (conversely, activate the doPick only if *varOwnerPicked* = *true*) and, when *varOwnerPicked* = *true*, doMoveTo*Room* causes both *varPositionOwner* and *varPositionNurse* to be set to the same *Room*. As an example of interaction between services, consider `kitchenService` and `pantryService`. As depicted in Figure 2, they do not have any action in common. Though, cooking any dish (namely, invoking doCook action on the `kitchenService` service) is *not* possible if some ingredients are missing (i.e., if *varIngredients* = *false*). The `pantryService` can buy them (indeed, doBuyIngredients sets *varIngredients* = *true*), but only after the execution of a check (doCheckIngredients). The evolution of doCheckIngredients is constrained by the *varIngredients* domain variable: if *varIngredients* = *false*, then the next state of `pantryService` is missing (and the doBuyIngredients action *executable*), otherwise it remains in the ready state.

Such comments motivate the advantages of decoupling services and dynamic domain as in the framework: the evolution of the system is not straightforward from the inspection of services or dynamic domains alone. Indeed, a service represents the behavior of a real device or application plugged in the environment, and it is distributed by vendors who do not know the actual context in which it will be used. The same service could affect (or be affected by) the world in different ways, according to the environment it is interacting with.

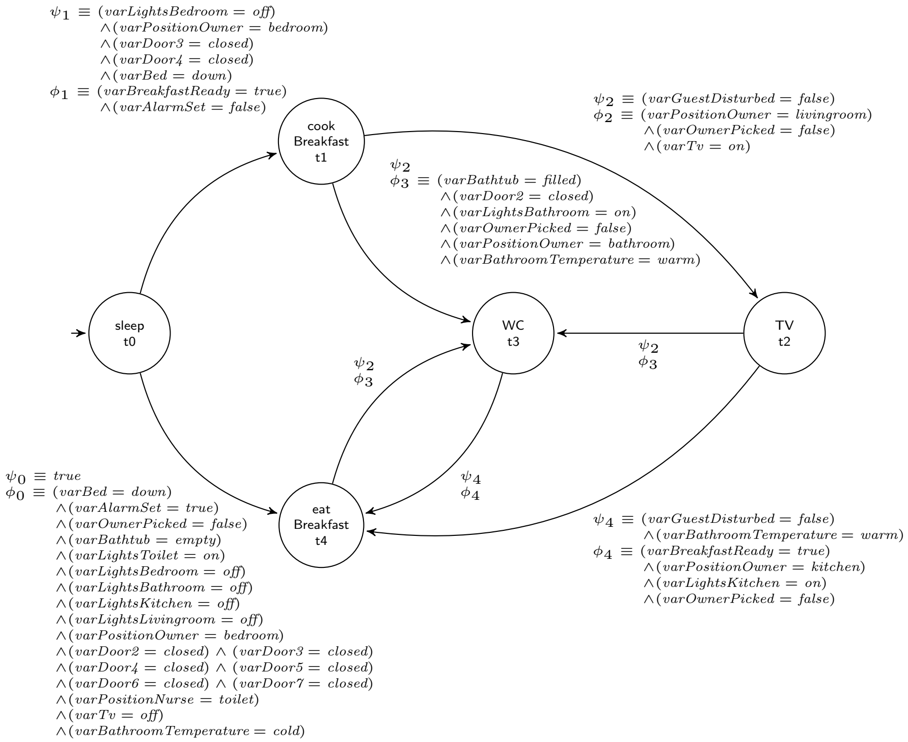

$\psi_1 \equiv (varLightsBedroom = off)$
$\wedge(varPositionOwner = bedroom)$
$\wedge(varDoor3 = closed)$
$\wedge(varDoor4 = closed)$
$\wedge(varBed = down)$
$\phi_1 \equiv (varBreakfastReady = true)$
$\wedge(varAlarmSet = false)$
cook Breakfast t1
$\psi_2 \equiv (varGuestDisturbed = false)$
$\phi_2 \equiv (varPositionOwner = livingroom)$
$\wedge(varOwnerPicked = false)$
$\wedge(varTv = on)$
ψ_2
$\phi_3 \equiv (varBathtub = filled)$
$\wedge(varDoor2 = closed)$
$\wedge(varLightsBathroom = on)$
$\wedge(varOwnerPicked = false)$
$\wedge(varPositionOwner = bathroom)$
$\wedge(varBathroomTemperature = warm)$
sleep t0
WC t3
TV t2
ψ_2
ϕ_3
ψ_2
ϕ_3
ψ_4
ϕ_4
$\psi_0 \equiv true$
$\phi_0 \equiv (varBed = down)$
$\wedge(varAlarmSet = true)$
$\wedge(varOwnerPicked = false)$
$\wedge(varBathtub = empty)$
$\wedge(varLightsToilet = on)$
$\wedge(varLightsBedroom = off)$
$\wedge(varLightsBathroom = off)$
$\wedge(varLightsKitchen = off)$
$\wedge(varLightsLivingroom = off)$
$\wedge(varPositionOwner = bedroom)$
$\wedge(varDoor2 = closed) \wedge (varDoor3 = closed)$
$\wedge(varDoor4 = closed) \wedge (varDoor5 = closed)$
$\wedge(varDoor6 = closed) \wedge (varDoor7 = closed)$
$\wedge(varPositionNurse = toilet)$
$\wedge(varTv = off)$
$\wedge(varBathroomTemperature = cold)$
eat Breakfast t4
$\psi_4 \equiv (varGuestDisturbed = false)$
$\wedge(varBathroomTemperature = warm)$
$\phi_4 \equiv (varBreakfastReady = true)$
$\wedge(varPositionOwner = kitchen)$
$\wedge(varLightsKitchen = on)$
$\wedge(varOwnerPicked = false)$

Fig. 3. The sample target process

Next we turn to the target process itself, shown in Figure 3, representing what Niels wants to happen, when waking up in the morning. First, the home system must let Niels get awaken only after the breakfast is ready: this is the aim of the first transition, where the reachability goal is to have $(varBreakfastReady = true) \wedge (varAlarmSet = false)$, while all conditions that make Niels sleep comfortable must be kept: $(varBed = down) \wedge (varLightsBedroom = off) \wedge \ldots$ Then, once the alarm rang out, we let Niels decide whether he prefers to have a bath or to watch TV (and optionally have a bath afterwards). In both cases, we do not want to wake up Dan ($\psi_2 \equiv (varGuestDisturbed = false)$). Niels can successively have breakfast, but we suppose that further he can go back to the bathroom and eat a little more again how many times he wishes:

this is the rationale beneath the formulation of: $\psi_4 \equiv (\mathit{varGuestDisturbed} = \mathit{false}) \land (\mathit{varBathroomTemperature} = \mathit{warm})$. Finally, Niels can get back to the bed room. The transition from the eatBreakfast (t4) state to the sleep (t0) one has no maintenance goal (i.e., $\phi_0 = \mathit{true}$), whereas the reachability goal is just to reset the domain variables to their initial setup.

4 Solver

As we can see from the case study above, goal-based processes can be used to naturally specify the behavior of complex long-running intelligent systems. In order to apply this framework to real applications, however, we need a practical and efficient solver for such composition tasks. The solution in [9], [10] reduces the composition problem to LTL synthesis by model checking. As a result, an efficient model checker and the approach is viable in practice only if large computational resources are available; on typical hardware for the smart home applications, only simple examples can be solved with that approach.

In light of the success of heuristic search in classical planning, it is interesting to ask whether this problem can be more efficiently solved by a direct search method. In this paper, we pursue this idea, and propose a novel solution to the composition problem based on an AND-OR search in the space of execution traces of incremental partial policies. Intuitively, the search keeps a partial policy at each step, and simulates its execution, taking into account all possibilities of the goal requests and the nondeterministic effects of the actions. If no action is specified for some situation yet, the policy is augmented by trying all possible actions for it. Starting from an empty policy, this process is repeated until either a valid policy is found that works in all contingencies, or all policy extensions are tried yet no solution is found. In this process, the nondeterministic goal requests and action effects are handled as "AND steps," whereas the free choice of actions during expansion represents an "OR step." Figure 4 shows the Prolog code of the body of our solver.

The composition starts from an empty policy [ ] with the initial goal state and initial world state S_0, and simulates (while incrementally building) its execution, until all possible goal requests in the target process can always be achieved by some policy C (line 2). In our implementation, we always assume that the intial goal state is 0. To handle all the possible evolutions of the system from goal state T and world state S, the `compose/4` [2] predicate first finds all goal requests GL that originate from T, and augments the current policy C_0 to obtain C_1 that handles these requests (lines 4–6). This is done by `compGoals/5`, which represents the first AND step in the search cycle. It recursively processes each goal request in the list GL by using `planForGoal/8` (lines 8–11). Notice that the policy is updated in each recursive step with the intermediate variable C in line 11. The predicate `planForGoal/8` essentially performs conditional planning with full observability and nondeterministic effects (lines 13–20). Lines 14 and

[2] According to the Prolog convention, we use the syntax <*name of the predicate*>/<*arity*>.

```
% planner.pl - a generic solver for goal-based process composition.
% Usage: call compose($C$) to find a realization $C$.
compose($C$) :- initial_state($S_0$), compose($0, S_0, [\,], C$).

% compose($T, S, C_0, C_1$) compose for goal state $T$.
compose($T, S, C_0, C_1$) :-
    findall($\langle M, G, T' \rangle$, goal($T, M, G, T'$), $GL$), !,
    compGoals($T, GL, S, C_0, C_1$).

% compGoals($T, GL, S, C_0, C_1$) compose for all goal requests in $GL$.
compGoals($\_, [\,], \_, C, C$).
compGoals($T, [\langle M, G, T' \rangle | GL], S, C_0, C_1$) :-
    planForGoal($T, M, G, T', S, [\,], C_0, C$),
    compGoals($T, GL, S, C, C_1$).

% planForGoal($T, M, G, T', S, H, C_0, C_1$)
% update policy for a specific goal.
planForGoal($\_, \_, \_, \_, S, H, \_, \_$) :- member($S, H$), !, fail.
planForGoal($\_, M, \_, \_, S, \_, \_, \_$) :- \+ holds($M, S$), !, fail.
planForGoal($T, \_, \_, T', S, \_, C, C$) :- member($\langle T, T', S, \_ \rangle, C$), !.
planForGoal($\_, \_, G, T, S, \_, C_0, C_1$) :- holds($G, S$), !,
compose($T, S, C_0, C_1$).
planForGoal($T, M, G, T', S, H, C_0, C_1$) :-
    bestAct($G, A, S$), next_states($S, A, SL$),
    tryStates($T, M, G, T', SL, [S|H], [\langle T, T', S, A \rangle | C_0], C_1$).

% tryStates($T, M, G, T', SL, H, C_0, C_1$)
% compose for all progressed world states.
tryStates($\_, \_, \_, \_, [\,], \_, C, C$).
tryStates($T, M, G, T', [S|SL], H, C_0, C_1$) :-
    planForGoal($T, M, G, T', S, H, C_0, C$),
    tryStates($T, M, G, T', SL, H, C, C_1$).
```

Fig. 4. Prolog implementation of our search-based solver

15 prevent the found partial policy from containing deadloops or violating the maintenance goal. Line 16 detects visited states in achieved goals so that they can be realized in the same way, and thus no further search is needed. Line 17 checks whether the current achievement goal has been realized, and if so, it goes on to recursively compose for the next goal state. Finally, Lines 18–20 capture the last case where no action is associated to the current situation, in which case the current policy needs expansion to handle it. This is done by the OR step of the search cycle, which proposes a best candidate action with the predicate `bestAct/3`, and planning goes on for the resulting world states.

Since the actions are nondeterministic, it means that executing an action may lead to multiple possible states, and the policy we find must work for them all. In our algorithm, `tryStates/8` handles all these states by recursing into `planForGoal/8` with updated policy for each state, which represents the second AND step in the search cycle (lines 22–25).

Recall that the exploration of a search branch may fail in Lines 14 and 15, due to a deadloop and violation of a maintenance goal, respectively. When either case occurs, the program backtracks to the most recent predicate with a different succeeding assignment, which is always `bestAct/3`. From there, the next best action is proposed and tried, and so on. If all the possible actions have been tried, yet none leads to a valid policy, the program backtracks to the next most recent `bestAct/3` instance, and the same process is performed similarly.

Notice that our algorithm is applicable to any goal-based process composition task, as the predicates `initial_state/1`, `holds/2`, `bestAct/3`, `next_states/3` and `goal/4` behave according to the actual target goal-based process and its underlying dynamic environment which are specified using a problem definition language detailed below. It is not hard to see that the our algorithm strategically enumerates all valid policies, generating *on-the-fly* action mappings for *reachable* situations only. The algorithm can be shown to be sound and complete.

Theorem 2 (Soundness and completeness). *Let $\mathcal{T}$ be a target goal-based process and $\mathcal{S}$ its underlying dynamic system. If* `compose(C)` *succeeds, then* `C` *is a realization of $\mathcal{T}$ in $\mathcal{S}$. Moreover, if $\mathcal{T}$ is realizable in $\mathcal{S}$, then* `compose(C)` *succeeds.*

This algorithm can be used for solving small composition problems even with a simple enumeration-based implementation of `bestAct/3`. However, as the problem size grows, this naïve implementation quickly becomes intractable, due to the large branching factor and deep search tree. Therefore, some intelligent ordering is needed for the succeeding bindings of `bestAct/3`, in order to make our solver efficient for large composition tasks. In our implementation of the solver[3],

[3] The code of both solver and case study, as well as the experimental results, are available at the URL: `http://www.dis.uniroma1.it/~cdc/pubs/CoopIS2012-Code_Tests.zip`

we make use of the well-known *delete-relaxation heuristics* [13], although other heuristics in classical planning could be adapted as well.

The delete-relaxation heuristics for a state is computed by solving a relaxed goal-reachability problem where all negative conditions and delete effects are eliminated from the original planning problem. It can be shown that the relaxed problem can always be solved (or proven unsolvable) in polynomial time. If a relaxed plan is found, then the number of actions in the plan is used as a heuristic estimation for the cost of achieving the goal from the current state; otherwise it is guaranteed that no plan exists to achieve the goal from the current state, so it is safe to prune this search branch and backtrack to other alternatives. In our implementation, when choosing the best action, `bestAct/3` first sorts all legal actions according to the optimistic goal distance of their successor states using the delete-relaxation heuristics[4], and unifies with each of the actions in ascending order when the predicate is (re-)evaluated. Notice that the heuristics only changes the ordering of branch exploration in the search tree, with possible sound pruning for deadends, and does not affect the correctness guarantee of our algorithm.

For our solver, a problem specification is a regular Prolog source file which contains the following components:

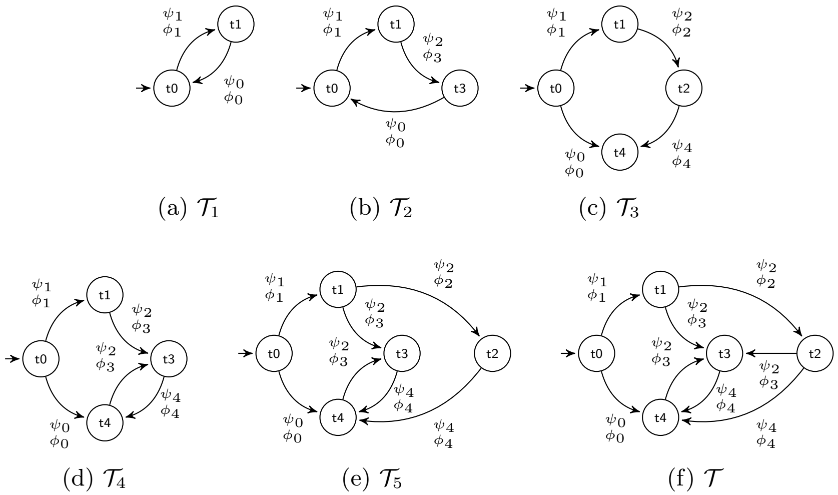
- the instruction to load the solver `:- include(planner).`
- a list of primitive fluents, each fluent `F` specified by `prim_fluent(F).`
- a list of primitive actions, each action `A` specified by `prim_action(A).`
- action preconditions, one for each action `A`, by `poss(A,P).` where `A` is the action, and `P` is its precondition formula.
- conditional effects of actions of the form `causes(A,F,V,C).` meaning that fluent `F` will take value `V` if action `A` is executed in a state where condition `C` holds.
- initial assignment of fluents of the form `init(F,V).` where `F` is the fluent and `V` is its initial value.
- the process by a list of goal transitions of the form `goal(T,M,G,T').` where `T` and `T'` are the source and target goal states of the transition, `M` is the maintanence goal, and `G` the achievement goal. By default, the initial goal state is always 0.

5 Experiments on the Case Study

In order to test the efficiency of the solution presented in Section 4, we conducted some experiments based on the case study of Section 3.

Given the dynamic system $\mathcal{S}$ and the target process $\mathcal{T}$ described in Section 3, we considered both $\mathcal{T}$ and its restrictions $\mathcal{T}_{i\in\{1,\ldots,5\}}$, shown in Figure 5, where states and goals refer to the ones depicted in Figure 3.

[4] In our experiments, we also take into account the conjunction of the maintenance goals of the next goal state, so that states violating future maintenance goals are pruned earlier, leading to further gain in efficiency.



(a) $\mathcal{T}_1$ (b) $\mathcal{T}_2$ (c) $\mathcal{T}_3$

(d) $\mathcal{T}_4$ (e) $\mathcal{T}_5$ (f) $\mathcal{T}$

Fig. 5. Test target processes

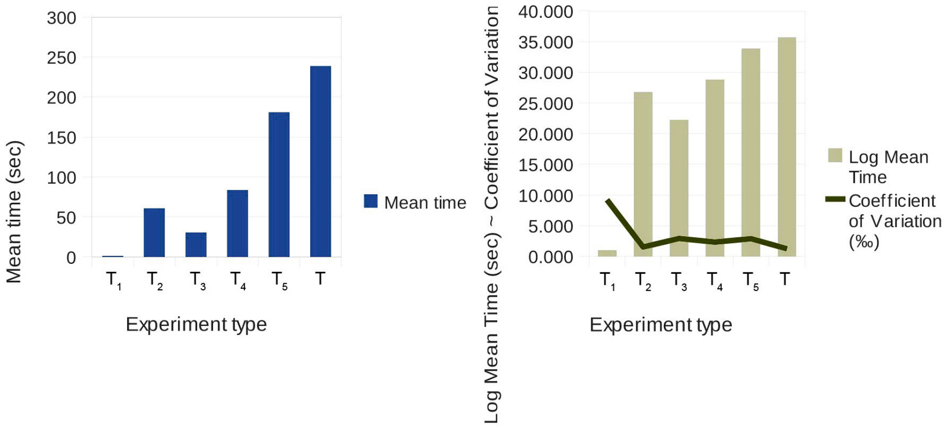
	Trans.'s	Time [sec]			Std. Dev.	Coeff. of Var. [‰]
		Mean ($\mathcal{M}$)	min (m)	Max (M)	(σ)	($\sigma/\mathcal{M}$)
$\mathcal{T}_1$	2	1.166	1.15	1.19	0.011	9.219
$\mathcal{T}_2$	3	60.688	60.54	60.82	0.094	1.545
$\mathcal{T}_3$	4	30.448	30.39	30.61	0.088	2.896
$\mathcal{T}_4$	5	83.497	83.26	83.88	0.195	2.331
$\mathcal{T}_5$	7	180.894	180.29	182.05	0.523	2.889
$\mathcal{T}$	8	238.841	238.38	239.19	0.291	1.219

Fig. 6. Test results

Hence, for each $\mathcal{T}_i$, we run the solver 10 times in a SWI-Prolog 5.10.2 environment, on top of an Intel Core Duo 1.66 GHz (2 GB DDR2 RAM, Ubuntu 10.04) laptop.

We gathered the results listed in Figure 6. The solution for the complete problem was found in about 239 seconds. Performances for simpler formulations followed an almost linear trend with respect to the input dimension, measured in terms of number of transitions in the target process (see Figure 7a). Figure 7b[5] shows that such results are quite reliable, since the Coefficient of Variation (i.e., the ratio between the Standard Deviation σ and the Mean Value $\mathcal{M}$) is fair little (ca. 2‰, excluding the first value, which is not significant, being the solution for that instance computed in too few milliseconds) and keeps constant as the $\mathcal{M}$ value grows. The performances are notable, especially if compared to the previous tests. Indeed, we ran a solver based on model checking techniques, built on top of TLV (version 4.18.4, see [20]), on the laptop mentioned above.

[5] There, the base for the Logarithm of the Mean Time $\mathcal{M}$ is the least $\mathcal{M}$ value.



(a) Tests results

(b) Coefficient of variation

Fig. 7. Graphical representation of the test results

Notice that such a solver requires the usage of high computational resources, which are not affordable in a smart home scenario. Indeed, on our laptop it took more than 24 hours to terminate, whereas the solver presented in this paper is returned a solution within less than 4 minutes.

6 Conclusions

In this paper we have proposed an approach for composing goal-based processes on the basis of available services, which stems from the real needs of a smart home scenario, in which available services are based on sensors, actuators and equipments of the home. The approach and the solver we develop turned out to be effective in practice as the case study and the experiments demonstrate. Future work include the investigation of the case of multiple users (e.g., all inhabitants of the home) asking for different simultaneous goal-based processes. Preliminary ideas can be found in [21,10].

Acknowledgements. This work has been partly supported by the EU projects SM4All (FP7-224332), ACSI (FP7-257593) and Greener Buildings (FP7-258888), and by the Italian AriSLA project Brindisys. Yuxiao Hu would like to thank his PhD supervisor Hector Levesque for useful discussions and financial support.

References

1. Baligand, F., Rivierre, N., Ledoux, T.: A Declarative Approach for QoS-Aware Web Service Compositions. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 422–428. Springer, Heidelberg (2007)

2. Beauche, S., Poizat, P.: Automated Service Composition with Adaptive Planning. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 530–537. Springer, Heidelberg (2008)
3. ter Beek, M.H., Bucchiarone, A., Gnesi, S.: Formal Methods for Service Composition. Annals of Mathematics, Computing and Teleinformatics 1(5), 1–10 (2007)
4. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic Service Composition based on Behavioural Descriptions. International Journal of Cooperative Information Systems 14(4), 333–376 (2005)
5. Blythe, J., Ambite, J. (eds.): Proc. of ICAPS 2004 Workshop on Planning and Scheduling for Web and Grid Services (2004)
6. Cardoso, J., Sheth, A.P.: Introduction to Semantic Web Services and Web Process Composition. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 1–13. Springer, Heidelberg (2005)
7. Catarci, T., Di Ciccio, C., Forte, V., Iacomussi, E., Mecella, M., Santucci, G., Tino, G.: Service Composition and Advanced User Interfaces in the Home of Tomorrow: The SM4All Approach. In: Gabrielli, S., Elias, D., Kahol, K. (eds.) AmBI-SYS 2011. LNICST, vol. 70, pp. 12–19. Springer, Heidelberg (2011)
8. Curbera, F., Sheth, A., Verma, K.: Services Oriented Architectures and Semantic Web Processes. In: ICWS 2004 (2004)
9. De Giacomo, G., Patrizi, F., Sardiña, S.: Agent Programming via Planning Programs. In: AAMAS 2010 (2010)
10. De Giacomo, G., Felli, P., Patrizi, F., Sardiña, S.: Two-player Game Structures for Generalized Planning and Agent Composition. In: AAAI 2010 (2010)
11. 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 Systems 11(10-12) (2011)
12. Hassen, R.R., Nourine, L., Toumani, F.: Protocol-Based Web Service Composition. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 38–53. Springer, Heidelberg (2008)
13. Hoffmann, J., Nebel, B.: The FF Planning System: Fast Plan Generation through Heuristic Search. Journal of Artificial Intelligence Research 14, 253–302 (2001)
14. Kaldeli, E., Warriach, E.U., Bresser, J., Lazovik, A., Aiello, M.: Interoperation, Composition and Simulation of Services at Home. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 167–181. Springer, Heidelberg (2010)
15. Klein, A., Ishikawa, F., Honiden, S.: Efficient QoS-Aware Service Composition with a Probabilistic Service Selection Policy. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 182–196. Springer, Heidelberg (2010)
16. McIlraith, S., Son, T.: Adapting GOLOG for Composition of Semantic Web Services. In: KR 2002 (2002)
17. Medjahed, B., Bouguettaya, A., Elmagarmid, A.: Composing Web Services on the Semantic Web. Very Large Data Base Journal 12(4), 333–351 (2003)
18. De Paoli, F., Lulli, G., Maurino, A.: Design of Quality-Based Composite Web Services. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 153–164. Springer, Heidelberg (2006)
19. Pistore, M., Marconi, A., Bertoli, P., Traverso, P.: Automated Composition of Web Services by Planning at the Knowledge Level. In: IJCAI 2005 (2005)
20. Pnueli, A., Shahar, E.: The TLV System and its Applications. Tech. rep., Department of Computer Science, Weizmann Institute, Rehovot, Israel (1996)

21. Sardiña, S., De Giacomo, G.: Realizing Multiple Autonomous Agents through Scheduling of Shared Devices. In: ICAPS 2008 (2008)
22. Schuller, D., Miede, A., Eckert, J., Lampe, U., Papageorgiou, A., Steinmetz, R.: QoS-Based Optimization of Service Compositions for Complex Workflows. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 641–648. Springer, Heidelberg (2010)
23. Wang, H., Zhou, X., Zhou, X., Liu, W., Li, W., Bouguettaya, A.: Adaptive Service Composition Based on Reinforcement Learning. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 92–107. Springer, Heidelberg (2010)
24. Wu, D., Parsia, B., Sirin, E., Hendler, J., Nau, D.S.: Automating DAML-S Web Services Composition Using SHOP2. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 195–210. Springer, Heidelberg (2003)
25. Yang, J., Papazoglou, M.: Service Components for Managing the Life-cycle of Service Compositions. Information Systems 29(2), 97–125 (2004)
26. Zhao, H., Doshi, P.: A Hierarchical Framework for Composing Nested Web Processes. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 116–128. Springer, Heidelberg (2006)