Integrating, Composing and Simulating Services at Home

Eirini Kaldeli, Ehsan Ullah Warriach, Jaap Bresser,
Alexander Lazovik and Marco Aiello*

Distributed Systems Group
Johann Bernoulli Institute
University of Groningen
Nijenborgh 9, 9747 AG, The Netherlands
{e.kaldeli,e.u.warriach,j.bresser,a.lazovik,m.aiello}@rug.nl

Abstract. Pervasive computing environments such as our future homes are the prototypical example of a dynamic, complex system where Service-Oriented Computing techniques will play an important role. A home equipped with heterogeneous devices, whose services and location constantly change, needs to behave as a coherent system supporting its inhabitants. In this paper, we present a fully implemented architecture for domotic applications which uses the concept of a service as its fundamental abstraction. The architecture distinguishes between a pervasive layer where devices and their basic internetworking live, and a composition layer where services can be dynamically composed as a reaction to user desires or home events. Next to the architecture, we also illustrate a visualization and simulation environment to test home coordination scenarios. From the technical point of view, the implementation uses UPnP as the basic device connection protocol and techniques from Artificial Intelligence planning for composing services at runtime.

Keywords: Pervasive Services, Internet of Things, Composition

1 Introduction

The vision of the Internet of Things brings a number of fresh challenges, that the field of Service-Oriented Computing can help to address. Having a large number of autonomous and heterogeneous objects whose location, connectivity, and set of functionalities may change during a home's life cycle, requires a rich and flexible infrastructure. Support for interoperation, dynamic discovery, sensing of the current execution context, and run-time service compositions are among the most notable elements of such an infrastructure.

In this paper, we focus our attention on the smart home. Following the vision of the Smart Homes for All project [19], we design and implement a software architecture based on the concept of service, that supports the integration of

* The research is supported by the EU project Smart Homes for All (http://www.sm4all-project.eu), contract FP7-224332.

heterogeneous home devices, the inference of the home context, and the possibility to compose services inside the home as a response to a user need or a home event. Technologies based on Service Orientation are not new for pervasive systems. UPnP and Jini [6] have been proposed as protocols and architectures for dynamic device and functionality discovery, based on describing services in terms of WSDL and Java interfaces respectively. These are excellent starting points for our study, as they provide support for basic interoperation, but to realise genuinely smart homes, more aspects need to be designed in terms of home sensing and composition.

Our approach is driven by the proposition that domestic events, may these be generated by a user's desire or by a home situation that needs to be handled, can be best addressed by designing a complex behavior specific to the event and the current home context. When a fire breaks for example, one does not simply want to turn on a fire alarm, but rather, based on what services are available in the home in terms of alarms, sprinklers, automatic doors, and so on, infer the status of the home and the location of the user, and then compose the available services to ensure maximum safety for the home inhabitants, as well as protection for the home itself. Such a philosophy of design for pervasive systems also brings an extra added value: the system is portable to several homes with minimal reconfiguration. In fact, the same event will be dealt with differently in different homes, simply because the available services will be diverse, as well as the state of the environment.

The paper makes concrete the vision and philosophy above by resorting to Artificial Intelligence (AI) planning techniques for service composition, and UPnP as the basic protocol for interoperation. Building on our previous work on service composition [12, 9] and creating a framework for integrating devices, we implement an instance of a SM4All architecture which is able to deal with physical and simulated devices, and also visualize home behaviors. The implementation is then evaluated to show that, despite the fact that we use elaborate AI techniques, the system performs rapidly, and a road to actual home deployment is definitely feasible.

The paper is organized as follows. A description of a possible scenario in a smart home, working as our running example, is presented in Section 2. Then we introduce the SM4All architecture, and we focus on the composition and pervasive layers in Section 3. Section 4 provides details of the RuG ViSi visualization tool. The results of performance evaluation for the framework at both its composition and pervasive layer are presented in Section 5. A discussion of related work and conclusions are presented in Sections 6 and 7 respectively.

2 Getting a beer

The soccer world-championship is well under way and the user of our smart home, as many others, likes to watch TV with a cold beer in his hands. Without too much planning, he simply has to make sure that there are beers available in

the house, and that he has paid the electricity bills so that the TV can work. This simple scenario can help illustrate the behavior of our smart home.

Let's assume that the inhabitant of the house has just taken his bath, and wants to move to the sitting room to watch the forthcoming football match. Such a request may include instructions about how the sitting room atmosphere should be prepared—by adjusting the lights, probably opening the window if the temperature is too high, and turning on the TV. During the halftime break, the user decides to go to the kitchen to prepare something to eat. While being there, the smoke detector in the kitchen identifies a potentially dangerous smoke leak——but fortunately not due to fire. As a result, a predefined home goal for dealing with this situation is automatically triggered: after having ensured that the user has safely moved out of the kitchen (let's say to the adjacent sitting room), the door leading to the kitchen is closed to isolate the smoke in a single room. The ventilator, if present, is turned on and the kitchen window is also opened, so that the foul air is expelled, while an alarm notification appears on the TV screen. While waiting in the sitting room, the user wants to move back to the kitchen, but only after having assured that the environment there is safe, and the smoke has been eliminated. This wish implies resorting to sensing to identify the current situation in the kitchen. Let's assume that after some time the smoke is indeed eliminated, causing the alarm on the TV and the ventilator to automatically turn off, and the user can finally move to the kitchen. After verifying that no serious damage has been caused, he decides to move back to the sitting room in order not to miss the second half of the match, that has just started.

While sitting on the sofa, and trying to overcome the stress from the unfortunate smoke leak incident, the user wishes to have a cold beer in his hand. Assuming that the household is equipped with a robot device, which is able to move around the house, get(put) items from(to) particular places and sense their temperature, the task for getting a cold beer can be assigned to the robot. Let's say that the user has neglected to put any beers in the fridge, however the system finds out that there are some beers left on the store shelf. Having this information in hand, the robot will move to the storage room and get a beer from there. In order to satisfy the requirement that the beer should be cold, it will proceed in placing the beer it has taken in the fridge, and leave it there for two minutes to cool. Then it will get it out again and bring it to the sofa. It should be noted that if the same goal was issued in another home instance, in which the robot device has only the capability of getting items from the fridge, the user would be unfortunate enough to be left without his highly desired cold beer, if there is no one such available in the fridge.

3 Architecture

The middleware is the software layer that abstracts from distribution, providing a coherent application interface. In the case of the Smart Home the middleware is a *thick* layer that has to offer a number of services to the participating com-

ponents. It has to accommodate for dynamic group membership, asynchronous communication, provide a common message ontology, support heterogeneous and mobile devices, mobility of the user. Most notably, it has to coordinate atomic device functionalities to satisfy elaborate application needs, that is, not simply expose a de-localized remote function, but rather aggregate and temporize existing functions in order to provide an added value complex functionality.

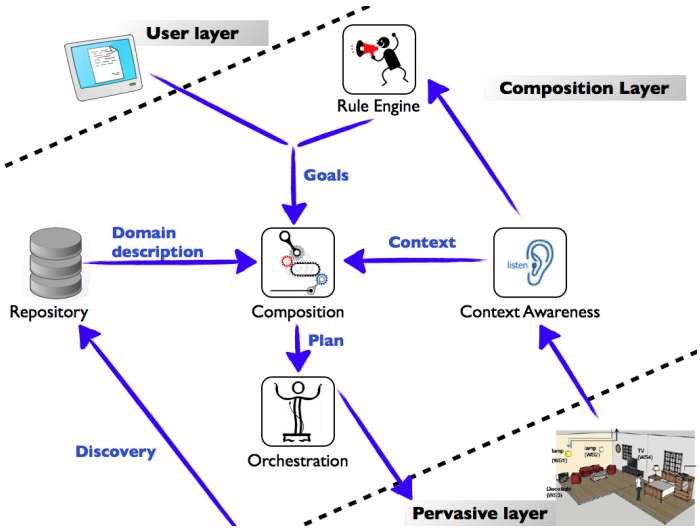

Fig. 1: Architectural overview.

In the context of the Smart Homes for All project, we propose a middleware architecture split into three macro layers. At the bottom is the *pervasive layer*, where the heterogeneous sensors, actuators and mobile devices of the house live. In the middle sits the *composition layer*, which is responsible for registering and inferring the state of the home, as well as coordinating it on behalf of the user. On top is the *user layer* which provides the interface to the home. This schematization is illustrated in Figure 1 with the main control and information flows represented by arrows. In this paper, we focus on the composition and pervasive layer, and provide an instance of the architecture. The user layer using touch screen and brain computer interface [8] is beyond the scope of the present treatment.

The central composition layer is further abstracted into five major components. The *context awareness* module is responsible for the collection of the sensed information from the home, and the maintenance of a representation of the execution and user context in the home, by reading information directly from the pervasive layer [5]. The *repository* keeps a number of key data bases which include a registry of description of abstract devices, a registry of currently active devices, semantic descriptions of service invocations, and the layout of the house

(e.g., the rooms that comprise it, and how they are arranged). A *rule engine* is constantly informed about any changes in the context, and identifies whether certain conditions hold. If the conditions entail that some action has to be taken, the rule engine directly invokes the composition module. The *composition* module is the central one of the composition layer. It is responsible for finding the right combination of service operations that can satisfy the high level complex goals, issued either by the rule engine (e.g., an emergency goal for combating some dangerous gas that has been identified) or by the user layer (e.g., a request for a beer). The composition module has to be aware about the home description stored in the repository, as well as about the current state of the environment, as seamlessly provided by the context awareness component. The working of the module is based on AI planning [7], therefore we shall interchangeably refer to it as the composition module or *planner*. Once a composition of services is computed, it needs to be executed. The execution is controlled by the *orchestration* module, which retrieves and invokes the physical services. Since the current state of the environment constantly changes, and these changes may interfere with the process in execution, the orchestrator should be able to use the feedback from each invocation to drive the rest of the execution.

In the context of the SM4All project, we have instantiated the general architecture described above using state-of-the-art and novel approaches we have developed. In particular, we use a constraint-based approach to planning in order to compose services [9], and UPnP [1] and OSGi [2] to provide a uniform infrastructure at the pervasive layer. In the followings, we describe in more details the characteristics of each component and how it functions.

Pervasive Layer The pervasive layer is a dynamic and open environment where devices join and leave while offering and consuming services. A number of requirements have to be satisfied. Firstly, new services should be automatically detected, and the interested parties should be notified accordingly. Secondly, the services should be described in a standardized programmatic manner, and it should be possible to control them in accordance with this description. Thirdly, interested parties should be notified about changes of services' states in a event-driven manner, and communication between services should be enabled regardless of the platform each service runs on. Moreover, the pervasive layer should be able to perform well with varying loads and number of participating devices.

To realize the layer and satisfy the above requirements, we use *Universal Plug and Play (UPnP)* [1] as the protocol for the direct access to hardware services, WSDL and SOAP protocols to expose high-level services, and the *OSGi framework* [2] as the intermediate between the physical UPnP and the WSDL-level service invocations. Figure 2 provides an overview of the architecture of the pervasive layer. At the bottom sits the *network layer* where physical devices can dock. UPnP devices use TCP/IP and UDP as basic networking protocols, but alternatives to UPnP, such as Bluetooth or ZigBee (`www.zigbee.org`), are also possible. OSGi is the framework for the wrapping of devices (UPnP or non-UPnP through the use of a proxy), providing a standard interface for repre-

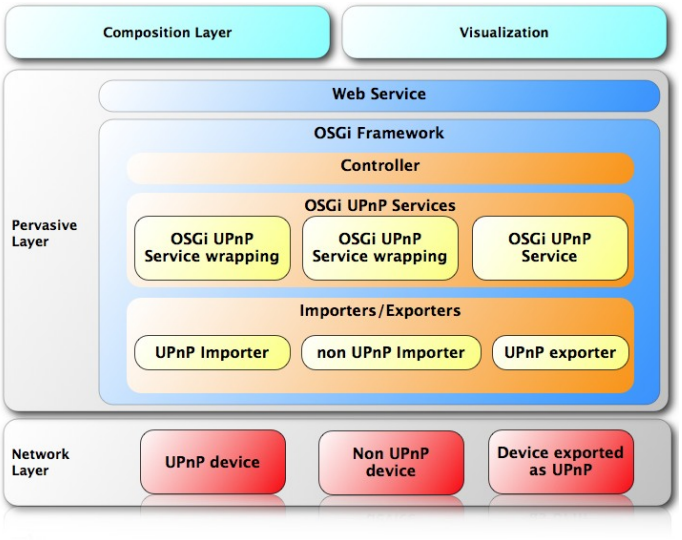

Fig. 2: Pervasive layer architecture.

sentation and interaction. Following the UPnP specification defined in the OSGi platform, a device includes a set of services, each of which supports some *actions*, and involves some *state variables*, which describe the current state of the service. All components in the OSGi framework are deployed as so-called bundles. The *Controller* is a special OSGi bundle that is responsible for handling events and controlling the services available in the framework, functioning as a bridge between the OSGi layer and the *Web Service (WS) layer*, which provides a standardized API to the upper layers. On top are the clients, which can invoke the services exposed by the available devices. Thanks to an event propagation mechanism, registered clients are notified when UPnP services become available or unavailable, and can subscribe to change events concerning the state variables they are interested in. The clients may be a visualization and simulation tool (see Section 4), a BPEL orchestration engine [13], or a composition layer.

The composition layer The planner is the module standing at the core of the composition layer. Its task is to compute a *plan*, that is, a sequence of actions that need to be applied in order to satisfy a given goal. Starting from an initial state, which in our case is reflected by the current values of all variables that describe the home domain, the application of each action in the plan leads to a new state, as prescribed by its effects. Referring to Figure 1, we go through the interactions that take place between the planner and the other components of the composition layer. The planner first retrieves the description of the home stored in the repository, and forms the planning domain, by mapping each UPnP action to a planning-level action, specified in terms of preconditions and effects

turn_on_ventilator
`Preconditions:`
$ventilator := OFF$
`Effects:`
$ventilator := ON$

move_robot(destination)
`Preconditions:`
$robotLocation \neq destination \wedge$
$(adjacent\_same\_room(robotLocation, destination) \vee$
$adjacent\_same\_room(robotLocation, destination) \wedge$
$door\_open(room(robotLocation), room(destination))$
`Effects:`
$robotLocation := destination$

Fig. 3: Examples of two planning-level actions.

(see Figure 3). This process has to take place once for each house instance, and be repeated only when a new service is discovered or removed.

The planner is being constantly informed about the current values of the variables describing the house by the context module, that receives the notifications about any changes in the environment in accordance with the event propagation mechanism mentioned in Section 3. Upon getting this information, the planner updates accordingly its initial state. Whenever a goal is issued, the planner performs a search to satisfy the goal under the conditions entailed by the specific home and the current initial state. Then the actual plan is generated, and passed further to the orchestrator, which maps each planning-level abstract action to the equivalent concrete UPnP action to be executed in the pervasive layer. In Figure 3 we provide two examples of actions, as described at the planner level. The first action *turn_on_ventilator* states that the action can be applied if the ventilator is OFF in the current state, and has as a result that it will be ON in the next state. The action *move_robot*(*destination*) instructs how the robot can move to a *destination*, provided as a parameter. The action can be applied if the *destination* does not coincide with the robot's current location *robotLocation*, and if either the current and the destination locations are adjacent to each other and belong to the same room, or, in the case they are neighbor locations but in different rooms, the door between these two rooms is open. Other actions, such as opening doors, can be applied to satisfy the preconditions of the *move_robot* action. This way, the moving will take place in steps, with the robot maneuvering between neighbor locations, based on how these are arranged in the specific house instance. Abiding by such a generic and loosely-coupled encoding, the actions that are common in all houses have to be specified once, without being tied to the details of each specific house.

The planner is a domain-independent CSP-based planner [9], which provides a number of features that are of particular relevance to the requirements associated with smart domotic environments. Firstly, it supports efficient handling of variables with large domains, which are frequently present in intelligent component interactions—e.g., temperature measurements, or the number of beers in our example. Moreover, current advances in the CSP field allow the employment of powerful inference and search techniques to speed-up the search. An important characteristic of the planner, that makes it especially well-suited for adaptable

and user-centric environments, is that it allows the expression and satisfaction of extended goals. The supported goal language accommodates for temporal constructs and maintainability properties, and adopts a clear distinction between sensing and achievement goals. The goal is expressed in a declarative way, i.e. it prescribes what properties should be satisfied and under which conditions, but not how the operations should be combined.

Goal 1: watch TV	achieve-maint($TvState = ON \ \wedge \ sitrLight1 = ON \wedge$ $sitrLight2 = OFF \ \wedge \ userLocation = AT\_SOFA) \wedge$ (achieve-maint($sitrWindow = OPEN$) under_condition_or_not(find_out-maint($sitrTemperature > 30$)))
Goal 2: address smoke leak (by Rule engine)	achieve-maint($kitchVentilator = ON \wedge$ $TvState = ALARM \ \wedge kitchWindow = OPEN) \wedge$ (achieve-final($doorsLeadTo(KITCHEN) = CLOSED$) under_condition(achieve-maint($personRoom \neq KITCHEN$)))
Goal 3: smoke eliminated (by Rule engine)	achieve-maint($kitchVentilator = OFF \ \wedge \ TV = OFF$)
Goal 4: go to kitchen if safe	achieve-maint($userLocation = AT\_OVEN$)) under_condition(find_out-maint($kitchSmoke = OFF$))
Goal 5: bring cold beer	achieve-final($robotLocation = userLocation \wedge$ $robotHolds = BEER \ \wedge \ beerTaken = COLD$)

Table 1: The goals corresponding to the test-case scenario

A set of predefined-defined goals can be made available to the user, hidden behind the buttons that appear in the user interface panel. If the goal issued can be satisfied, the generated plan is executed and the UPnP devices change state accordingly. If the goal is not satisfiable under the current context, a message is shown on the user interface. Table 1 summarizes the goals that correspond to the scenario informally described in Section 2. The syntax and semantics of most constructs can be found in [9] (the language has been enriched with a couple of constructs thereafter). In the case of Goal 1, the `under_condition_or_not` structure ensures that the subgoal $sitrWindow = OPEN$ will be satisfied if the temperature is higher than 30 degrees, while if the temperature is lower than that, then only the rest of the subgoals will be looked after. This is to be contrasted with the semantics of the `under_condition` construct, which in Goal 4 for example dictates that $kitchSmoke = OFF$ should necessarily hold, otherwise the goal will fail. The `find_out` type of subgoals take care of sensing. In the case of `achieve-final` subgoals, the respective proposition has to be

satisfied at the final state, but is allowed to hold or not throughout the plan execution, like for example in Goal 5 where *robotLocation* will change many times while the robot is moving around to find and get the beers. On the other hand, the `maint` annotation implies that once the proposition is satisfied, it should remain true in all subsequent states, preventing the variables involved to change many times in the plan states traversal.

It is worth noting that the user does not have to know about the operational details of the service instances available in each specific house. It is up to the planner to find a "creative" solution based on the capabilities of the particular house and the current context, without depending on any ad-hoc business processes. Another useful feature is that, thanks to a mechanism of dynamic addition and removal of constraints, continual planning for newly-issued goals can be performed in an effectual way, as well as the incorporation of recent changes reported by the context-awareness component, after removing the obsolete information.

4 Simulation and Visualization

Testing and verifying the behavior of service orchestration in large pervasive systems is a costly and error prone enterprise, which demands a vast amounts of time and effort. Therefore, an environment that mimics as closely as possible the real setting, and is able to simulate a number of interactions and behaviors can greatly help the development and testing of service-based pervasive computing applications. Following our initial implementation of a visualization tool for home environments—the RuG ViSi tool—based on Google SketchUp [13], we have upgraded it to a module compliant with the SM4All architecture, which is capable of full bidirectional interactions with UPnP services or devices deployed with a UPnP proxy. The visualization module is registered as a client of the server on top of the OSGi framework. This way, one can control the devices at the pervasive layer through their virtual equivalents, while the invocations of the actual devices are in turn reflected at the visualization layer, by changing its state. Figure 4 depicts a screenshot from the RuG ViSi tool, with a virtual house (in the center) surrounded by a number of UPnP devices: a door controller, a wall-fan, a light, two window controllers, a fire alarm and a television set. A UPnP module which represents the position of a user in the house, and can be coupled with a location detector that provides information about the position of the user, is also there (at the bottom). Real physical devices, that may follow different standards and communication protocols, such as Bluetooth or ZigBee, can also be linked to the RuG ViSi tool. For instance, we have coupled a light service with a Sentilla mote (`www.sentilla.com`) equipped with an accelerometer and a radio connection. The hardware is plugged in the OSGi layer and, when shaken, turns on and off a specific light in the virtual home.[1]

[1] A demo is available at `www.youtube.com/watch?v=2w_UIwRqtBY`.

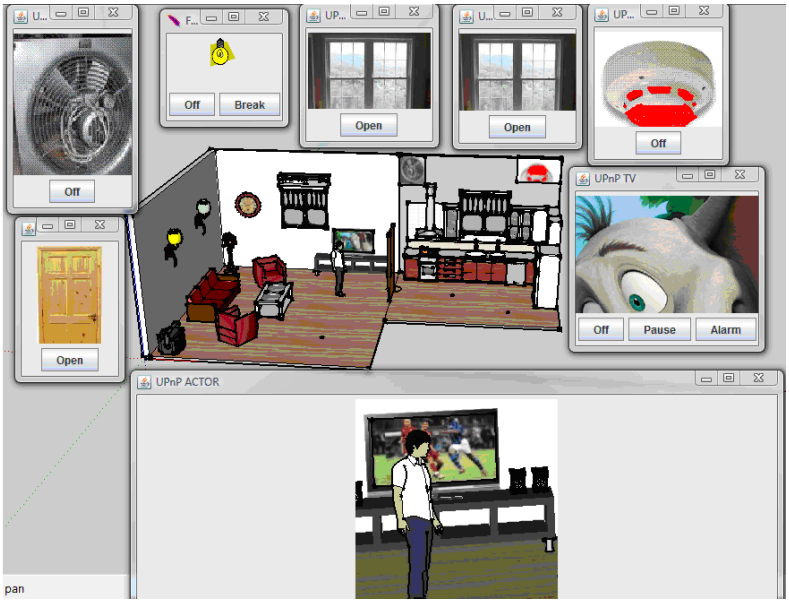

Fig. 4: Simulated UPnP devices in the home.

5 Evaluation

We have implemented the architecture described in Section 3 and the simulation and visualization tool described in Section 4. Next, we provide an evaluation of our implementation to show the viability of a home solution based on such an architecture. We start by evaluating the pervasive layer, and then look into the composition layer and consider its interactions with the other modules in the system.

Evaluating the pervasive layer. First, we test the latency in the pervasive layer. The setup is based on using a 2.66 Ghz computer running Windows 7, 64 bit and Java 1.6.0_18. The devices used for the test are implemented as OSGi bundles, i.e., the devices are simulated and wrapped in OSGi. Every device has one service which has a state variable and action. The clients are implemented in Java using Apache CXF 2.2.5. For the purposes of the evaluation, we define the following quantities and measures: *t_stimulus* is the time at the beginning of the invocation of a UPnP action of one of the test devices, and *t_response* the time a client receives a notification that a state variable changed as result of the invocation of an action. *Latency* is the absolute temporal difference between stimulus and response to an action.This way, latency includes both service control latency (latency related to the invoking of the action) and eventing latency (latency related to the event notification mechanism).

In the context of this test, a device is a simulated OSGi UPnP device, with one service, one state variable *count* of type integer, and one action called *setCount*, which acts as a setter for *count*. Since each device has one service, the terms (UPnP) device and (UPnP) service are used interchangeably. A client is an instance of a WS client whose only function is to record the time, when it is informed about state variable changes. Real clients, such as the visualization client or the composition layer, would of course include more functionalities.

The testing protocol is as follows. After bootstrapping, each client subscribes to *device_count* state variables (of *device_count* different devices). Then, the following step is repeated *iteration_count* times, with increasing values of *current_iteration*: for each device *t_stimulus* is stored, its action is invoked with *current_iteration* as a value, and then a sleeping time of *sleep_time* ms is issued. Upon receiving a notification that a state variable has changed, each client stores the current time in the *t_response* which correpsonds to the specific state variable and iteration (as reflected by the received value *current_iteration*). Finally, the time measurements are aggregated to compute latency. Figure 5 shows how the performance of the system behaves when the number of clients increases, by plotting the average latency. As one can see, for up to 30 clients the latency is well within acceptable bounds (20ms) and then grows very fast. We also tried to increase the number of devices up to 2.000 and have experienced latency times in the order of 6 ms. The evaluation indicates that the tool can support many clients and a high number of devices, still providing very low response times. Next, we consider a very special type of client: the composition module based on planning.

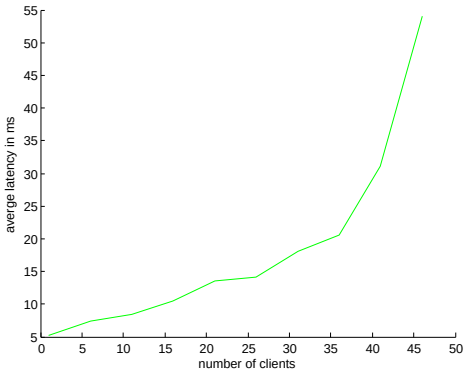

Fig. 5: Average latency of iteration_count = 10, device_count = 100, sleep_time = 100ms

Evaluating the composition Layer Considering the scenario of Section 2, we implement all devices and services according to our architecture, and provide an initial evaluation of the performance of the composition layer. The tests have been run on a 1.83 Ghz computer running Debian lenny, 32 bit and Java 1.6.0_12. The components are described with respect to the OSGI UPnP Device Specification and are exposed as OSGi bundles, with each device supporting one or more services, each of which involves a number of actions and state variables. The constraint solver standing at the core of the planner is the Choco v2.1.1 constraint solving library (`www.emn.fr/x-info/choco-solver`). The composition layer is registered as a client to the WS server of the middleware, and subscribes to all services comprising the domain. In the specific

evaluation, we model a home with 5 rooms, and 10 devices providing 21 UPnP actions (plus the sensing operations that are defined for each state variable in the domain), which affect 22 different state variables. The user himself is represented as one of the services at the pervasive layer. It should be noted that a state variable can be accessed or affected by more than one services, possibly belonging to different devices, like the $FridgeDoor$ variable, which, besides the $Fridge$ device, can also be directly set by the *Robot*, when for instance it wants to cool a beer. Each of the exposed UPnP actions is mapped to an equivalent planner-level action, as the ones shown in Figure 3.

We have tested the planner on each of the goals in Table 1, for different initial states in accordance with our testing scenario. Table 2 shows two examples of such plans. Each plan is represented as a partially ordered set of actions, with comma-separated actions $a1, a2$ indicating that a_1 has to be performed before a_2, while the actions included in the same set $\{a_1, \ldots, a_n\}$ can be executed in parallel. As one can see, the same goal for getting a cold beer results in different plans, depending on the current context: if there are no beers in the fridge (`[5a]`) the robot has first to move to the storage to get a beer from there, and then cool it by moving back to the fridge, while in the second case (`[5b]`) it can save some effort by getting the requested beer directly from the fridge. It should be noted that because of the use of random search strategies, the plans returned may slightly vary between different runs.

Goal/Initial state	Plan
`[5a]` Goal 5 (get cold beer) with initial state : *robotLocation=AT_START,* *userLocation=AT_SOFA,* *kitchStorDoor=CLOSED,* *sitrKitchDoor = CLOSED,* *numOfBeersInFridge=0,* *numOfBeersInStorage=8,* *robotHolds=EMPTY,* *fridgeDoor=CLOSED*	*open_fridgeDoor,* {*open_sitrKitchDoor,* *open_kitchStorDoor*}, *move_robot_to(AT_OVEN),* *move_robot_to(AT_STOR_SHELF),* *robotGetsBeerFromStorage,* *move_robot_to(AT_FRIDGE),* *robotCoolsBeer,* {*open_fridgeDoor,* *close_kitchStorDoor*}, *robotGetsBeerFromFridge,* {*move_robot_to(AT_SOFA),* *close_fridgeDoor*}
`[5b]` Same as above but with $numOfBeersInFridge = 1$	*open_fridgeDoor, open_sitrKitchDoor,* *move_robot_to(AT_FRIDGE),* {*robotGetsBeerFromFridge,* *open_kitchStorDoor, open_bedrBathrDoor,* *open_sitrBedrDoor*}, {*move_robot_to(AT_SOFA),* *close_fridgeDoor*}

Table 2: The plans generated for Goal 5 (getting a cold beer), for two different initial states (only the initial values that are of interest to the goal are mentioned) .

Test	Number of actions in plan	Plan Composition (time in sec.)	Plan execution (time in sec.)
`[1]` (watch TV)	10	1.5	1.1
`[2]` (address smoke leak)	9	1.1	0.8
`[3]` (smoke eliminated)	2	0.7	0.3
`[4a]` (go to kitchen, smoke on)	0	0.1	–
`[4b]` (go to kitchen, smoke off)	4–5	0.7	0.4
`[5a]` (get beer, fridge empty)	12–15	2.4	0.6
`[5b]` (get beer, fridge full)	6–9	2.1	0.5

Table 3: The time required for composition and execution . In [5a], which includes the *CoolBeer* action, we have subtracted the time the robot waits for the beer to cool.

The time required by the planner to subscribe to the available UPnP services, build the planning-level domain description, and sense the first initial state, by invoking the UPnP sensing actions for all state variables, is 9.7 sec. This is the 'home bootstrap' time and needs to be executed only once per house and per set of devices. We have measured the time the planner takes to generate a plan for each of the goals, for a given initial state, as well as the time needed for each plan to be actually executed by invoking the respective simulated UPnP actions. These results are summarized in Table 3, along with the number of actions included in the respective plan. We have used a random branching strategy during constraint solving, by restarting the search after a maximum number of backtracks. The reported times both for composition and execution are averaged over 5 separate runs. As already mentioned, the plans may differ in some of the test situations, in which cases we mention both the minimum and the maximum number of actions in the produced plans. It turns out that the most demanding goal is 5 (getting a cold beer), especially in the case where there are no beers already stored in the fridge, mainly due to the substantial backtracking required to find a solution (up to 478 backtracks, compared to 47 backtracks in the worst case concerning the other goals). In general, the more indirect the inter-relations between the different actions required to satisfy the goal are, the more search and backtracks are needed to compute the desired plan.

The changes entailed by the generated plan can also be visualized in the simulated home environment. The visualization client has been tested on a more restricted modelling of a house consisting of two rooms (see footnote 1 in Section 4).

6 Related work

In [4, 3] we survey domotic standards and propose to use the Web service stack as a mean to solve the interoperability problem at home. We show how WS-Notification can be used as an even-based mechanism for addressing emergency

situation in the home, most notably, the fall of the elder. The basic architecture is an eventing one with no notion of context and coordination of service beyond basic action/reaction interactions. The issue of composing domotic components has been addressed in [18], where composite services are deployed as BPEL processes, which are made available in a semantically enriched OSGi platform. These BPEL processes are predefined and not created at run-time based on sensed information. A more dynamic approach inspired from AI techniques is adopted in [17], where the problem of service integration is cast to Distributed Constraint Optimization. This is a highly distributed framework, however it suffers from a inflexible and cumbersome domain modelling process, while the requests the user can make are restricted to a set of rather simple commands that involve only a limited number of devices. AI planning techniques for Web Service composition have been proposed by a number of authors, e.g., [12, 11, 14]. A common denominator of most of the approaches in this area is that they can support a restricted variety of composite functionalities, either because they rely on—to a less or greater extent—fixed templates of pre-anticipated user behaviors, or because they support simple goals, with limited expressive power.

From the pervasive layer perspective, Service Oriented Architectures have been widely proposed, e.g., UPnP or Jini [6]. A richer form of "pervasive SOA" is proposed in [15], where the importance for home networks with platform independence and loose coupling is advocated. In [15] the challenges that currently exist in interconnecting home devices are described, and it is recognized that OSGi can be useful for developing smart homes. In [16], the semantic annotation of OSGi description is proposed to improve the discovery process. Looking at UPnP [1], its use as low level home middleware has been often proposed, e.g., [10].

7 Concluding remarks

Service-Oriented computing provides an advanced approach to building dynamic systems. If its initial thrust came from the need of integration of business information systems, the future may add a new important area: pervasive computing with our homes being an important instance. We have designed, implemented and evaluated a generic SOA for homes which supports highly dynamic computing context. Our initial evaluation indicates that the approach using AI planning, context awareness, and OSGi/UPnP device wrapping is a viable one.

To achieve a robust, scalable and user-friendly solution, many research challenges remain open. Dealing with *concurrency* and possible contradictions that may arise when events interfere with the execution of a plan, is an important extension of our framework. Improving the *efficiency* of the planner used for composition, and moving towards generating optimal plans, is also high in our agenda, as are *context* updates and efficiency in sensing. Another direction of future work involves further automating the process of transforming the pervasive-level services to planning-level actions by using an ontology that provides the

necessary semantic annotations. Security, privacy, and user interfacing are also important topics currently investigated by other partners of the SM4All project.

References

1. Upnp^{TM} device architecture version 1.1 (2008), `www.upnp.org`
2. OSGi service platform core specification release 4 (2009), `www.osgi.org`
3. Aiello, M.: The Role of Web Services at Home. In: IEEE Web Service-based Systems and Applications (WEBSA) (2006)
4. Aiello, M., Dustdar, S.: A domotic infrastructure based on the web service stack. Pervasive and Mobile Computing 4(4), 506–525 (2008)
5. Baldoni, R., Cerocchi, A., Lodi, G., Montanari, L., Querzoni, L.: Designing highly available repositories for heterogeneous sensor data in open home automation systems. In: Int. Ws. on Software Technologies for Embedded and Ubiquitous Systems. pp. 144–155. Springer (2009)
6. Dobrev, P., Famolari, D., Kurzke, C., Miller, B.A.: Device and service discovery in home networks with osgi. Communications Magazine, IEEE 40(8), 86–92 (2002)
7. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann, Amsterdam (2004)
8. Guger, C., Daban, S., Sellers, E. Holzner, C., Krausz, G. Carabalona, R., Gramatica, F., Edlinger, G.: How many people are able to control a P300-based brain-computer interface (BCI)? Neuroscience Letters 462, 94–98 (2009)
9. Kaldeli, E., Lazovik, A., Aiello, M.: Extended goals for composing services. In: Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009. AAAI (2009)
10. Kim, D.S., Lee, J.M., Kwon, W.H., Yuh, I.K.: Design and implementation of home network systems using upnp middleware for networked appliances. IEEE Transactions on Consumer Electronics pp. 963–972 (2002)
11. Kuter, U., Sirin, E., Nau, D., Parsia, B., Hendler, J.: Information Gathering During Planning for Web Service Composition. In: Journal of Web Semantics (2004)
12. Lazovik, A., Aiello, M., Papazoglou, M.: Planning and monitoring the execution of web service requests. In: ICSOC'03. pp. 335–350. LNCS 2910, Springer (2003)
13. Lazovik, E., den Dulk, P., de Groote, M., Lazovik, A., Aiello, M.: Services inside the smart home: A simulation and visualization tool. In: Int. Conf. on Service-Oriented Computing. LNCS, vol. 5900, pp. 651–652. Springer (2009)
14. Martínez, E., Lespérance, Y.: Web Service Composition as a Planning Task: Experiments using Knowledge-Based Planning. In: Proc. of the Workshop on Planning and Scheduling for Web and Grid Services (ICAPS-2004) (2004)
15. Ngo, L.: Service-oriented architecture for home networks. In: Seminar on Internetworking. pp. 1–6 (2007)
16. Panagiotis Gouvas, T.B., Mentzas, G.: An OSGi-Based Semantic Service-Oriented Device Architecture. In: OTM '07. pp. 773–782 (2007)
17. Pecora, F., Cesta, A.: DCOP for Smart Homes: a Case Study. Computational Intelligence 23(4), 395–419 (2007)
18. Redondo, R.P.D., Vilas, A.F., Cabrer, M.R., Arias, J.J.P., Duque, J.G., Solla, A.G.: Enhancing residential gateways: A semantic OSGi platform. IEEE Intelligent Systems 23(1), 32–40 (2008)
19. SM4All: Smart hoMes for All. http://www.sm4art-project.eu (2008-2011)