Automatic Description of Context-Altering Services through Observational Learning*

Katharina Rasch[1], Fei Li[2], Sanjin Sehic[2],
Rassul Ayani[1], and Schahram Dustdar[2]

[1] KTH Royal Institute of Technology
School of Information and Communication Technology
Stockholm, Sweden
{krasch,ayani}@kth.se
[2] Distributed Systems Group,
Vienna University of Technology, Austria
A-1040 Wien, Argentinierstrasse 8/184-1
lastname@infosys.tuwien.ac.at

Abstract. Understanding the effect of pervasive services on user context is critical to many context-aware applications. Detailed descriptions of context-altering services are necessary, and manually adapting them to the local environment is a tedious and error-prone process. We present a method for automatically providing service descriptions by observing and learning from the behavior of a service with respect to its environment. By applying machine learning techniques on the observed behavior, our algorithms produce high quality localized service descriptions. In a series of experiments we show that our approach, which can be easily plugged into existing architectures, facilitates context-awareness without the need for manually added service descriptions.

1 Introduction

Context covers all aspects of the current situation of a user, for example user location, current time, physical properties (e.g. temperature, humidity) or medical data (e.g. heart rate and blood pressure). Often the services offered by pervasive devices are what we call *context-altering*, i.e. executing them changes the user context in some way. For instance, an air conditioner service influences the indoor temperature, and switching on a stereo changes the noise level around the user. In our prototype smart home we found that of the 30 available services, over 90% are context altering.

Context-altering services can be described with a context-dependent precondition and effect. The precondition describes the situations under which the service can be executed; the effect of the service describes the context changes that are typically induced when executing the service. Knowledge of the preconditions and effects of context-altering services allows the system to react to

* This work is supported by EU FP7 STREP Project SM4ALL (Smart hoMes for ALL), under Grant No. 224332.

J. Kay et al. (Eds.): Pervasive 2012, LNCS 7319, pp. 461–477, 2012.

undesired user situations. For example, the system may use the knowledge of user preferences regarding indoor temperature, humidity et cetera to regulate these properties using the available pervasive devices. This view has been taken up in [10]: based on the current user situation, the system automatically composes and executes the services that are necessary to change the context according to the user requests. In [13] we propose a proactive service discovery approach that continuously recommends the currently most interesting services based on user preferences.

Generally the service descriptions should be as detailed as possible, to allow the system to select the best fitting action in any given situation. However, the service descriptions can not be provided a priori by the device manufacturer, since they are highly dependent on the local environment. For instance, opening a window on a cold day will change the readings of the nearby temperature sensors; however, information about which sensors are nearby is not available to the manufacturer, and may even be situation-dependent. For this reason, service descriptions have to be manually fitted to the environment, such that they reflect the interdependencies between the installed devices and sensors. The descriptions must also be updated whenever new sensors and devices are added to the system. However, the manual description of services is tedious and error-prone, hindering non-expert users to adopt the technology [15]. Having to rely instead on technicians to install new devices is not acceptable for most users.

We propose in this paper a method for automatically learning the capabilities of context-altering services without the need for manual input by users or technicians. The idea is to observe the behavior of the service with respect to its environment. Clustering and classification are then applied to the observed data to find typical patterns of preconditions and effects. The proposed approach produces localized service descriptions that reflect the setup of the environment and support situation-dependent effects. Using our approach, new pervasive devices can simply be plugged into an existing installation without manual configuration. Instead the system will automatically recognize the capabilities of the new services and can use them, for example, in service recommendation and composition.

Our contributions in this paper are threefold: we *(1)* extend a formal model for context and context-altering services [13,16] (Section 2), *(2)* present novel algorithms for automatically learning service capabilities from the service behavior using clustering and classification (Section 3) and *(3)* propose an architecture for integrating capability learning in pervasive systems (Section 4). The capability learning is evaluated in a series of experiments in Section 5. The paper finishes with an inspection of the related work and our conclusions.

2 Preliminaries

Before we can proceed with describing our approach for learning service capabilities, it is first necessary to formalize the notions of context and context-aware services.

Context. In [13,16] we present a context model that formalizes the connection between context and context-altering services. The Hyperspace Analogue to Context (HAC) models context as a multidimensional space, where each dimension denotes a type of context and describes the data type (nominal or numeric) and range of the context information. All dimensions together span the multi-dimensional space of all possible context descriptions. The current context of a user can be described as a point in the context space, for example the context point $c = \langle d_{location} = kitchen, d_{curtains} = closed, d_{temp} = 25 \rangle$ describes that the user is currently in the kitchen where curtains are closed and the temperature is 25 °C. If the user moves to the bedroom, then a context change $\Delta c = \langle d_{location} = bedroom \rangle$ can be observed.

We consider the internal status of a device a part of context, since it plays an intrinsic role for describing the user situation. For example, assume that the user is waking up in the morning in a dark room. The different actions the system can take to move the user into a more comfortable awake situation depend on the internal status of the devices in the room; only if the curtains are closed at the moment, opening them is a viable option for increasing the brightness in the room, otherwise the installed lamps must be used.

Context-Altering Services. The idea of a service with context-dependent preconditions and effects is similar to the world-altering services described in [17]; though since we model the surroundings of a service as context, we prefer the more unassuming term *context-altering* for these services. The precondition of a service can be used to filter out unavailable services – only if the current user context fulfills the condition, then this service is a possible execution choice. The service effect describes how the execution of the service changes the user context, and is important for identifying if a service can change an undesired user situation to a more comfortable one.

This service definition does however not consider that many services in pervasive environments can have situation-dependent side effects. For instance, opening the window if it is cold outside will quite likely decrease the indoor temperature; analogously if it is hot outside, the indoor temperature will increase. Service side effects capture how outside factors, for example seasonal changes, influence service capabilities. Knowledge about service side effects can help identify alternatives for broken or undesired services, e.g. opening the window if it is cool outside could be an energy-efficient alternative to using the air conditioner. Definition 1 extends the previous notion of context-aware services [13] with service side effects.

Definition 1 (Context-altering service). *A context-altering service is situated in HAC and is described with:*

- s^{pre}*, the main precondition of the service; when* $c \in s^{pre}$ *then* s *is a possible execution choice.*
- s^{eff}*, the main effect of the service; executing* s *will change* c *according to* s^{eff}*.*

– s^{side}, *the set of side effects of the service. In* $s^{side} = \{(s^{spre_0} \rightarrow s^{seff_0}), \ldots, (s^{spre_m} \rightarrow s^{seff_m})\}$ *each pair* $(s^{spre_i} \rightarrow s^{seff_i})$ *with* $0 \leq i \leq m$ *describes one side effect of the service. If* $c \in s^{pre}$ *and* $c \in s^{spre_i}$*, then executing* s *will change* c *according to* s^{eff} *and* s^{seff_i}*.*

Consider an example service s for opening a window: due to mechanical restrictions, the window can only be opened if the curtains in front of it are also open, and opening the window changes not only the internal status of the device, but may also have effects on the room temperature. Service precondition and effect are straightforward to describe: s^{pre}=$\langle D_{window} = closed, D_{curtain} = open\rangle$ and s^{eff}=$\langle D_{window} = open\rangle$.

Each side effect of a service is paired with an additional precondition that describes the context situation under which the side effect occurs. The actual effect of s is then a combination of the service's main effect and all side effects that can be fulfilled. The window service may have a side effect with $s^{spre_0} = \langle D_{tempIn} = [19 - 22], D_{tempOut} = [22 - 30]\rangle$ and $s^{seff_0} = \langle D_{tempIn} = [22 - 30]\rangle$, describing that opening the window on a warm day will increase the indoor temperature. Several such side effects for varying outdoor temperatures may be described for this service.

The distinction between main preconditions/effects and side effects highlights the importance of the former for service discovery and composition. First main preconditions have to be checked to see whether a service is available in a given situation. Only if this is the case, then side effects can provide useful additional knowledge.

3 Learning Service Capabilities

Knowledge of service capabilities allows a pervasive system to react to undesired context situations by executing or recommending services that can change the context to a more comfortable one. However service capabilities can often not be provided directly by device manufacturers. Only those parts of service preconditions and effects that concern a device's internal status can be described a priori. More expressive capabilities depend on the other devices and sensors installed in the pervasive environment, and are localized in several ways:

Preconditions. Inter-dependencies between several devices, often of mechanical nature. For example, curtain in front of window or small hallway were only one of the adjacent doors can be open at a time.

Effects. Depends on installed sensors. For example, heater or lamp cause changes in nearby temperature and light sensors. Several such sensors may be available, with different readings depending on their distance from the device.

Requiring either a technician or an experienced user whenever a new device is added to the system is simply not practical. Instead we propose to automatically learn capabilities by observing a service's behavior. The observations are collected in the service's execution history, and machine learning techniques are

applied to find precondition and effect patterns. The effect of a service can be identified by finding the context changes that commonly occur after its execution. Similarly the preconditions of a service are those conditions under which the service is commonly invoked. We start in the following by describing the service execution history, and how it can be obtained. Afterwards we present three algorithms for learning a service's precondition, effect, and side effects, respectively.

3.1 Service Execution History

Definition 2 formalizes the notion of a service's execution history: for every execution of a service, the context prior to the execution and the observed context change after the execution are recorded.

Definition 2 (Execution history). *The execution history H_s of a service s after k executions of s is a set $H_s = \{(c_0 \rightarrow \Delta c_0), \ldots, (c_{k-1} \rightarrow \Delta c_{k-1})\}$. In H_s each pair $c_i \rightarrow \Delta c_i$ with $0 \leq i < k$ records the current context c_i before the $i-$th execution of s and the context change Δc_i observed after the i-th execution of s.*

The set $H_s^{pre} = \{c_0, \ldots, c_{k-1}\}$ is called the precondition history and the set $H_s^{eff} = \{\Delta c_0, \ldots, \Delta c_{k-1}\}$ the effect history of s. As a simple example consider Table 1, documenting seven executions of a service s for opening window blinds.

The left hand side (a) of the table shows in seven rows the context c_i before each of the executions i. There is no connection between rows, i.e. other services may have been executed and changed c between two executions of s. The context is described using five different dimensions, including numeric and nominal dimensions. It stands out that the window blinds are always closed before the execution of s, whereas all other context dimensions vary. This observation indicates that $d_{blinds} = closed$ is a precondition of opening the blinds.

Table 1. Sample execution history

(a) Context before execution

	Temp	**Light out**	**Blinds**	**Light in**	**TV**
c_0	22	bright	closed	dark	off
c_1	21	dark	closed	dark	on
c_2	22	dark	closed	bright	off
c_3	20	bright	closed	dark	on
c_4	22	dark	closed	bright	on
c_5	22	bright	closed	dark	off
c_6	22	bright	closed	dark	off

(b) Changes after execution

	Blinds	**Light in**	**TV**
Δc_0	open	bright	-
Δc_1	open	-	off
Δc_2	open	-	-
Δc_3	open	bright	-
Δc_4	open	-	-
Δc_5	open	bright	-
Δc_6	open	bright	-

The right hand side (b) of Table 1 shows in corresponding rows the context changes Δc_i observed after the $i-$th execution of s. One clear difference between (a) and (b) is that in the former all five dimensions are used for describing c_i,

but in the latter only a subset of dimensions is used for Δc_i and many values are missing. This is because c_i describes the full context of the environment, containing values for each available context dimension. On the contrary, Δc_i contains information only about those dimensions whose value changed after the execution.

In the effect history (b) it stands out again that for each execution a context change is observed on the "Blinds" dimension, indicating that $d_{blinds} = open$ is an effect of opening the blinds. Context changes on the "Light in" dimension are only sometimes observed, hinting that $d_{lightIn} = bright$ could be a possible side effect of s. And indeed, when reviewing the precondition history (a), it can be seen that this effect always occurs if $d_{lightOut} = bright$ and $d_{lightIn} = dark$, i.e. a side effect of s is: if it is dark inside and bright outside, then opening the window blinds will result in it being bright inside. The single change on the "TV" dimension on the other hand must be regarded as an error as long as no further observations occur. Such an observation error can happen, for example, if the TV was turned off at the same time as the blinds were opened and the context change for the TV service was erroneously attributed to the blinds service.

Recording the Execution History. A major challenge with recording the execution history is to determine the optimal time for sampling context changes. The optimal *sampling time* is very much dependent on the context type that is being observed. Turning on a light is typically fast and a change in luminosity can be observed after a few seconds. However it may take 10 minutes or longer until significant temperature differences can be observed after turning on the heating. No a priori information is available about the latency of context changes.

A solution to this problem is to record all context changes after a service execution up to a very long sampling time. However, the longer the sampling time is, the higher is also the probability that other services are executed during this time and that the context changes caused by multiple services are overlapping. The optimal sampling time is thus a trade-off between missing important context changes and observing irrelevant changes. We will show later that our algorithms are strong enough to be able to deal with irrelevant context changes, thereby allowing the use of long sampling times which can avoid the problem of high-latency dimensions.

3.2 Learning Main Effects

The main effects of a service are constituted by those context changes which commonly occur when the service is executed. The changes may not necessarily occur in all service executions, since due to e.g. network errors some changes may be missing from the effect history. Therefore we consider a context change to be part of the main effect of a service, if it occurs in at least $\lfloor \vartheta_e * k \rfloor$ of all k service executions. For instance a ϑ_e of 0.9 ignores occasional missing context changes, while at the same time filtering out any unrelated context changes from simultaneously executed services.

Algorithm 1 for learning service main effects takes as input the effect history and the number of service executions, with a global constant ϑ_e. Starting from an empty service effect, for all dimensions that occur in at least one Δc_i of the history (Line 2-3) do: If d_i is nominal, find the most occurring nominal value v. If v occurs in enough service executions then set dimension i in the main effect to v (Line 4-8). If d_i is numeric, an outlier detection [4] must first be performed to remove all values that deviate markedly from the rest of the values on d_i. Such outlier values can occur if the simultaneous execution of other services also resulted in context changes on d_i. If after outlier removal enough values remain, then set dimension i in the main effect to the interval from the minimum value to the maximum value on d_i (Line 9-14).

Algorithm 1. Learning service main effects

```
1: procedure LearnMainEffects(H_s^eff, k)
2:     s^eff = ⟨⟩
3:     for ∀d_i ∈ D(H_s^eff) do
4:         if d_i is nominal then
5:             v = maxOccur(H_s^eff, d_i)
6:             if count(H_s^eff | d_i = v) ≥ ⌊ϑ_e * k⌋ then
7:                 s_i^eff = v
8:             end if
9:         else
10:            H'_s^eff = removeOutliers(H_s^eff, d_i)
11:            if count(H'_s^eff | d_i not missing) ≥ ⌊ϑ_e * k⌋ then
12:                s_i^eff = [min(H'_s^eff, d_i), max(H'_s^eff, d_i)]
13:            end if
14:        end if
15:    end for
16:    return s^eff
17: end procedure
```

3.3 Learning Main Preconditions

The main preconditions of a service are those context conditions that commonly hold when the service is executed. Analogously to the main effects, we say that a context point is a precondition of a service, if it holds for at least $\lfloor \vartheta_p * k \rfloor$ of all k service executions. Typically we set also $\vartheta_p = 0.9$. Algorithm 2 for learning preconditions proceeds similarly to learning main effects by looking at all dimensions occurring in the precondition history (Line 3). Nominal dimensions are handled analogously to Algorithm 1 (Line 4-5).

For numeric dimensions an additional step is performed to avoid too wide dimension intervals. Take for example a service for switching on a light which has been executed under varying conditions of the outside temperature $tOut$, such that the found interval is $s_{tOut}^{eff} = [-20, 50]$. This result has little informative

value as a service precondition, since it merely describes that the service can be executed under any conditions for $tOut$. As preconditions we instead want specific conditions described by a restricted value interval. Let $min(H^{pre}, d_i)$ be the global minimal value and $max(H^{eff}, d_i)$ the global maximal value on d_i in the precondition histories of any service. Then an interval on d_i is only considered a service precondition if it considerably restricts the dimension according to a parameter τ (Line 7-9). For example if $\tau = 0.5$, then only those intervals are considered that restrict the dimension to maximal half of its global size.

Algorithm 2. Learning service main preconditions

```
1: procedure LearnMainPreconditions(H_s^pre, k)
2:     s^pre = ⟨⟩
3:     for ∀d_i ∈ D(H_s^pre) do
4:         if d_i is nominal then
5:             ▷ proceed analogously to LearnMainEffects
6:         else
7:             if (max(H_s^pre, d_i) − min(H_s^pre, d_i)) / (max(H^pre, d_i) − min(H^pre, d_i)) < τ then
8:                 s_i^pre = [min(H_s^pre, d_i), max(H_s^pre, d_i)]
9:             end if
10:        end if
11:    end for
12:    return s^pre
13: end procedure
```

A final post-processing step must be performed after the preconditions of all services have been learned. It may be that the context contains static dimensions, which never or rarely change during the whole execution history, e.g. the internal status of an unused device. These dimensions provide little informative value as preconditions and thus should be excluded. If a dimension is static, then references to it will be contained in the preconditions of a majority of services. Therefore it needs to be checked for each dimension $d_i \in D(H^{pre})$, whether more than φ service preconditions contain the same value or value interval for d_i. If this is the case, then d_i is removed from the preconditions of all services.

3.4 Learning Side Effects

Side effects of a service are any additional precondition and effect pairs that can be found in the execution history. For the blinds service in Table 1, one could, for example, see the pair "If $d_{lightOut} = bright \wedge d_{lightIn} = dark$, then $d_{lightIn} = bright$". To be considered a side effect, a precondition and effect pair should be contained in at least $\lfloor \vartheta_s * k \rfloor$ and at most $\lfloor \vartheta_e * k \rfloor$ executions. Setting ϑ_s to a very small value increases the risk that context changes from parallel service executions are found to be side effects. A high ϑ_s on the other hand means that very rare side effects can not be found. For discovery and composition the

correctness of service descriptions is essential, therefore we typically set $\vartheta_s = 0.1$, i.e. only side effects occurring in at least 10% of service executions are considered.

As a preprocessing step it is necessary to remove from the history those dimensions which were already used in main preconditions and effects respectively or were found to be static. For preconditions, if $s_0^{pre} = v_1$, then there can be no side effect precondition $s_0^{spre} = v_2$, since side effects only occur if $s^{pre}$and s^{spre} hold at the same time, which is not possible in this case. Therefore for the precondition history $H_s'^{pre} = H_s^{pre} - D(s^{pre}) - D_{static}$. Analogously for the effect history $H_s'^{eff} = H_s^{eff} - D(s^{eff})$. Additionally, numerical dimensions are discretized to improve the quality of the learning result.

Finding Candidate Effects. It is much easier to find re-occurring patterns in the effect history than in the precondition history, since the former typically contains less data dimension. A pattern can also be called a cluster, denoted as a pair (D, O), where D is the set of dimensions and O the set of indexes of the objects forming this cluster. For the example data in Table 1b, a cluster would be found with $D = \{d_{lightIn}\}$ and $O = \{0, 3, 5, 6\}$. The minimum number of objects in such a cluster must be $minSupport = \lfloor \vartheta_s * k \rfloor$ for a service with k executions. For finding clusters we apply the DBSCAN clustering algorithm [6] and obtain a set of candidate effect clusters A_s, which will be the input to the next step.

Finding Matching Preconditions. Candidate clusters found do not necessarily describe actual service side effects. It can also happen that artificial clusters are found, which contain noise objects from simultaneous service executions, such as the TV effect seen in one execution of the example blind service. We consider a candidate cluster to describe a service side effect if a precondition can be found such that: if and only if the precondition is fulfilled, the side effect happens.

This requirement is very similar to the classification rules found by classification algorithms. Classification is a supervised data mining technique which is applied to an already classified set of training data to find rules for classifying unclassified data. By setting for each c_i in $H_s'^{pre}$ an additional class to indicate whether the item i is contained in the candidate cluster $a \in A_s$ or not, we can apply classification learning to $H_s'^{pre}$ in order to find any rules (i.e. preconditions) for this class (i.e. effect).

Algorithm 3 for side effects learning takes as input the precondition and the effect history of a service, with the main precondition and effect dimensions already removed, plus the number of service executions. In Line 3 effect clustering using DBSCAN is performed, producing a list of candidate clusters. For each of those candidate clusters each item c_i in the precondition history is annotated with a class `cluster` if i is contained in the cluster and a class `nocluster+i` otherwise (Line 5-9). A different class for each non-cluster item is used, since we want to avoid finding rules for $class =$ `nocluster`.

We have found that the PART classification rule learner [7] achieves good results on our context data. The result of running PART on the annotated $H_s'^{pre}$

Algorithm 3. Learning service side effects

```
1: procedure LearnSideEffects(H'_s^pre, H'_s^eff, k)
2:     s^side = ∅
3:     A = cluster(H'^eff, ⌊ϑ_s * k⌋)
4:     for ∀a = (D, O) ∈ A do
5:         for ∀c_i ∈ H'^pre do
6:             if i ∈ O then classes[i] = cluster
7:             else classes[i] = nocluster+i
8:             end if
9:         end for
10:        r = best(PART(H'^pre, classes) | r.class = cluster)
11:        if r.correct / (r.correct + r.incorrect) > β then
12:            s^side = s^side ∪ (toScope(r) → toScope(a))
13:        end if
14:    end for
15:    return s^side
16: end procedure
```

is a set of rules where each rule r is described by its condition, class and accuracy in terms of correct and incorrect classifications. We are only interested in the one rule r with $r.class = \texttt{cluster}$ with the highest number of correct classifications (Line 10). If this rule also has a low number of incorrect classifications according to a parameter β, then we can consider the pair $(r \rightarrow a)$ a side effect of the service (Line 11). Typically we set $\beta = 0.9$.

In Line 12 a function *toScope* is called, which converts rules and clusters to context scopes. For converting a cluster starting from an empty scope s^{seff} do for each dimension $d_i \in D$: if d_i is nominal, find the cluster value v that all items contained in the cluster take on d_i and set $s_i^{seff} = v$. If d_i is numeric, find the minimum value v_{min} and the maximum value v_{max} for all cluster items on d_i and set $s_i^{seff} = [v_{min}, v_{max}]$. Rules consist of a conjunction of $d_i = v$ (for nominal d_i) and $d_i = [v_{min}, v_{max}]$ (for numeric d_i), so converting a rule to a s^{spre} is straightforward.

4 System Architecture

We have implemented capability learning for the use in a smart home prototype. The software architecture of the smart home was deployed during the SM4All project [5]. Only the introduction of a dedicated *Capability learning* component and small changes in the user interface were needed to add automatic service capability learning to the existing pervasive architecture, as shown in Figure 1.

The pervasive environment, depicted in the lower part of the figure, contains the devices and sensors using Universal plug and play (UPnP [9]) and Konnex (KNX [1]) technologies. In order to be usable for our approach, devices and sensors must make their internal status and sensed data available to the system. UPnP directly fulfills this requirement; a device's internal status and sensed data

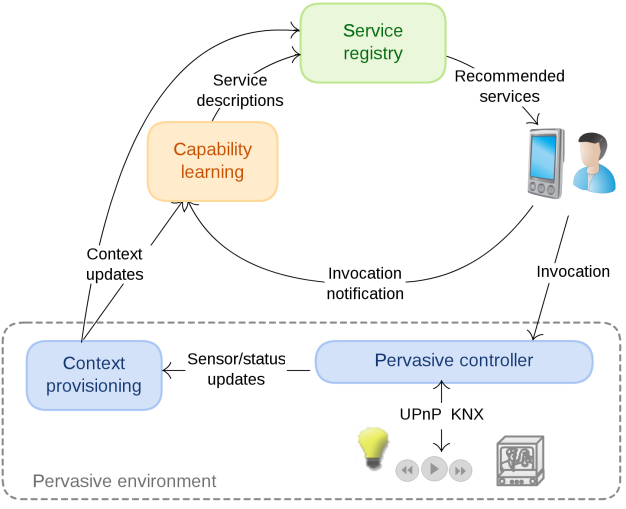

Fig. 1. System architecture

is published using the GENA (General Event Notification Architecture) framework. KNX on the other hand provides only the communication infrastructure to send control signals to devices and does not fulfill any of the requirements. To enable the use of KNX devices in the smart home, the *Pervasive controller* maintains and publishes the status of all installed KNX devices.

Context information from the various sources is collected and processed by the *Context provisioning* component. It is realized using the COPAL (COntext Provisioning for ALl) framework [14], a complex context processing system which produces formatted context events. The context information is used by the *Service Registry* to automatically generate service recommendations based on the current user context and preferences. In the original setup, the *Service registry* uses manually defined service descriptions for generating recommendations.

We have added the *Capability capturing* component, which implements the learning approach presented in this paper. The component is informed by the user client about which services were executed and listens to context change events to build the execution histories of the installed services. It then performs the capability learning, and sends any found service description to the *Service registry*, which can than be used for service recommendation.

5 Experiments and Results

This section reports the results of number of experiments which we carried out in the prototype smart home and through simulation using synthetic data.

5.1 Experiments in the Smart Home Prototype

Experimental Setup. The prototype is owned by Fondazione Santa Lucia[1] and is situated in Rome, Italy. The two bedroom flat is equipped with 13 pervasive devices and 7 sensors. The prototype is aimed mainly at users with physical disabilities, resulting in a strong focus on services for home control. Services are available for controlling doors, windows, curtains, lamps, an automatic bed and media devices. We found that of the 30 installed services, 28 are context-altering. Only two software services used for gathering input from the user are information-providing services [17]. The latency of context changes differs greatly between the devices. Turning on one of the lamps is fast, with a reaction time of about 30 milliseconds. In contrast, when raising or lowering the bed it takes 21 seconds until the action is finished and the device publishes an updated internal status.

Since the prototype is built into a research facility without any access to the outside world, it was not possible to observe any side effects from outside factors such as day/night changes. Most of the services have thus rather simple capabilities referencing only one dimension. Only two service preconditions make use of more than one dimension: the window curtain in the living room can only be closed if the window is closed, and vice versa the window can only be opened if the curtain is open.

For the experiment we set the sampling time to 25 seconds, to ensure that even the context changes from the slowest device were observed and used for capability learning. The parameters of the learning algorithms were set as follows: $\vartheta_e = 0.9$, $\vartheta_p = 0.9$, $\vartheta_s = 0.1$, $\tau = 0.5$, $\varphi = 0.3$, $\beta = 0.9$.

Metrics. In order to check the correctness of the learned capabilities, we first manually described all services, and compared the automatically learned descriptions with the manually provided ones. The number of executions that are necessary until the capabilities are learned correctly serves as a metric for evaluating how quickly the system learns.

Results. We started executing services in the home sequentially, i.e. with at least 25 seconds between each service execution. We found that under these circumstances the system learns the correct service capabilities with on average about 13 executions per service. Using these automatically generated descriptions, service recommendation worked just as well as with manually provided service descriptions. Due to a limited access to the prototype and the time-consuming nature of the experiments, we were only able to run this first experiment in the physical smart home. For further evaluating the system, we ran the following experiments in a simulation, using the same setup as the actual prototype, including devices, sensors and device latencies.

In real-world scenarios it is quite unrealistic that all services are executed sequentially. In order to evaluate how the learning approaches can deal with

[1] http://www.hsantalucia.it/

overlapping executions of several services, we gradually decreased the average time between service executions.

The services were executed in a random fashion, and each experiment was run for 10 replications. Table 2 lists the learning results for execution intervals ranging from 30 seconds (i.e sequential execution) to 1 second. In the latter case, 24 other services are executed during the sampling time of one service, creating heavily overlapping context changes. The number of necessary executions until the system correctly learns the capabilities are very stable for the 30, 10 and 5 seconds execution intervals. Only for the somewhat unrealistic case of the user continuously executing a service every second, does the number of necessary execution increase slightly.

Table 2. Number of necessary executions in prototype system

Average time between executions (in s)	30	10	5	1
Average necessary executions per service	13.1	13.8	14.4	20.8
(Standard deviation)	1.8	2.0	1.5	3.5
Worst case necessary executions	21	25	26.2	47.9

Table 2 also lists the worst case of necessary executions until the system correctly learns the capabilities. In these cases typically the effect was learned correctly, however the preconditions were temporarily too strict, e.g. the precondition for switching on the bedroom lamp correctly referenced the status of the lamp, but also required that the TV should be off. This behavior can be avoided by increasing the minimum support necessary for starting the learning process. The higher the minimum support, the longer it takes until all service descriptions are learned. With a lower minimum support, service descriptions are learned faster but may, in rare cases, be temporarily too strict. This system parameter can be easily tweaked by the user to achieve a preferred behavior.

5.2 Experiments Using Synthetic Data

Experimental Setup. The aim of the following experiments is to evaluate capability learning in a pervasive environment with a higher number of services and dimensions, plus more complicated preconditions, effects and side effects, as compared to the smaller-scale prototype system. The simulation environment creates to this end 70 dimensions in a mix of 50% nominal and 50% numeric dimensions. The dimension latencies are Gaussian distributed, with $\mu = \sigma^2 = 1$, i.e. most dimension values change between 1 or 2 seconds after service execution. 50 different services are created, each having main preconditions and effects, plus up to two two side effects. For creating a service effect, first up to three dimensions are randomly drawn from the set of all generated context dimensions, and a random value or value interval is selected for each of them and set as a

service effect. Generation of main precondition as well as side effects and their preconditions is performed analogously.

The execution of services is simulated, with the time between two service executions drawn from an exponential distribution with $\mu = executionInterval$. All context changes up to the preset *samplingTime* are recorded. The same parameters as in the prototype experiments were used for the learning algorithms. All of the experiments were performed on a desktop PC with an Intel Core 2 Duo CPU with 3 GHz and 4 GB RAM running Linux, and were run for 25 replications.

Metrics. The results of capability learning and the actual service capabilities are compared using the standard metrics of precision, recall and F1-score. Recall measures which percentage of the service capabilities of interest were found by the learning algorithm. Precision measures which percentage of the learned service capabilities of interest are actual capabilities. The F1-score is the harmonic mean of precision and recall, i.e. $F1 = 2 * (precision * recall)/(precision + recall)$.

Results. The first experiment tests, how well the capability learning algorithm works depending on the number of executions per service. To test the algorithm without any simultaneous service executions, we set $samplingTime = 10$ and $executionInterval = 20$.

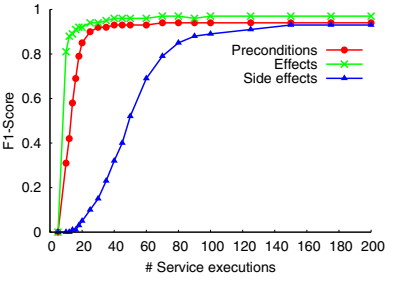

Fig. 2. Accuracy vs execution number

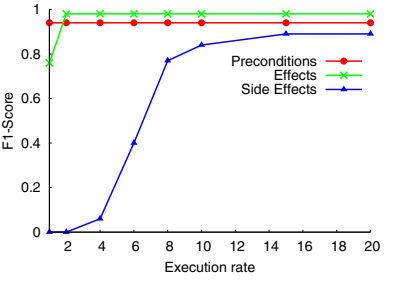

Fig. 3. Accuracy vs execution rate

It can be seen in Figure 2 that main preconditions and effects of a service stabilize very quickly with very high accuracy; after 25 executions of each service both precondition and effect learning achieve $F1 > 0.95$ (with 0.01 standard deviation). The learning of service effects stabilized slightly faster, which is not surprising since only a few context dimensions have to be mined, compared to all 70 dimensions for learning preconditions. The accuracy of side effects stabilizes much slower, since they only occur for some service executions, so more executions are necessary before side effects can be reliably mined. The accuracy strongly increases between 20 and 80 executions per service and stabilizes to $F1 > 0.9$ at around 125 executions (with 0.02 standard deviation).

Next we stress tested the algorithm with overlapping service executions. We set the number of executions per service to 100, keep $samplingTime = 10$ and

vary the *executionInterval*. For example for $executionInterval = 1$, service executions will happen on average every second, so for each of its executions a service will sample the context changes induced by itself and 9 other services. It can be seen in Figure 3 that the results for main preconditions and effects are very stable even for small execution intervals. The side effect accuracy is stable for two parallel service executions, but degrades for small execution intervals because context changes from overlapping executions are identified as side effects.

Finally we evaluated how well the algorithm can deal with a wide spread of dimension latencies by using a $samplingTime = 50$ and gradually increasing the variance of the generated latencies $1 \leq \sigma^2 \leq 40$. We found that the accuracy of the service capabilities is stable over the whole range of dimension latencies. In terms of algorithm runtime we found that capabilities can be learned in less than 2 seconds per service, for $executionInterval = 1$ and 100 executions per service.

The experimental results show that our approach is able to reliably and quickly learn service main preconditions and effects even under very stressed conditions. Side effect learning is most reliable under normal, more relaxed conditions where individual services are executed mostly sequentially.

6 Related Work

Compared to the number of proposed service description and annotation languages, research on automatic or semi-automatic service annotation is sparse. Patil et al. were among the first to point out the problems of manual annotation [15]. They present the METEOR-S web service annotation framework, which graphically assists the user in annotating web service descriptions. METEOR-S uses linguistic and structural matching between a service's functional description and candidate ontologies to identify the most fitting concepts for describing a service semantically and suggests them to the user during the annotation process. ASSAM is a similar tool by Heß et al., with the difference that ASSAM employs machine-learning techniques to identify fitting concepts by learning from already annotated services [8].

Bowers et al. exploit the additional knowledge contained in scientific workflows for annotation purposes [2]. Scientific workflow systems aim to integrate preprocessing of data, statistical, or data mining processes on the data and postprocessing and visualization of the results. An actor in such a workflow is any component (e.g. shell script, web service) that performs work on the data. Actors are annotated with information about the process they perform for facilitating automatic composition of such workflows. Bowers et al. propose to automatically infer missing annotations based on the connections between actors, e.g. if it is known that an actor produces descriptions of genes, then it can be inferred that any other actor that takes the descriptions as input performs work on such descriptions. A similar approach is taken in [11] for finding annotations for inputs and outputs of web services in a service workflow.

A number of works propose to execute services in order to learn about their functionalities. Lerman at al. use machine learning techniques to identify potential input data types for a service [12] and validate them by executing the service

with sample data and observing whether the output is satisfying or erroneous. Carman at al. aim to find out how the functionality of a service can be described in terms of other, already annotated services [3]. The target service and varying combinations of existing services are executed using the same input data, and the similarity of the output data is checked until a satisfying combination is found.

Our work distinguishes itself substantially from the described approaches. Previous works concentrate on finding descriptions for service category and input and output data types. As far as we know, no work has been published concerning the automatic description of context-altering services, which are common in pervasive environments. Additionally, all other approaches rely on specific information which may itself be hard to provide; typically either high-quality functional service descriptions and ontologies or partially annotated services and workflows are needed. In contrast, our approach relies solely on observing the environment. Only the data published by devices and sensors is needed, which is a functionality already built into popular technologies such as UPnP.

7 Conclusions and Future Work

In this paper we have presented a novel solution for automatically describing context-altering services through observational learning. Based on a formal model of context called HAC we have described a set of algorithms for learning service capabilities by mining on a service's past behavior. We have shown how service capability learning can be added to existing pervasive architectures. Experimental results, both in a smart home system and using simulation, demonstrate that our algorithms are able to reliably and efficiently identify service capabilities under varying conditions. Our approach is therefore a step towards plug and play context-awareness in pervasive environments.

In the future we want to extend our approach with more complicated relations between preconditions and effect that are currently not supported by our model, such as for example "If it is cooler outside than inside, then opening the window will make it cooler inside". It would also be interesting to study how service parameters, e.g. the setting of the air conditioner, influence service capabilities.

References

1. KNX standard (Version 1.1). Konnex Association Brussels (2004)
2. Bowers, S., Ludäscher, B.: Towards Automatic Generation of Semantic Types in Scientific Workflows. In: Dean, M., Guo, Y., Jun, W., Kaschek, R., Krishnaswamy, S., Pan, Z., Sheng, Q.Z. (eds.) WISE 2005 Workshops. LNCS, vol. 3807, pp. 207–216. Springer, Heidelberg (2005)
3. Carman, M.J., Knoblock, C.A.: Learning semantic descriptions of web information sources. In: Proceedings of the 20th International Joint Conference on Artifical Intelligence, pp. 2695–2700. Morgan Kaufmann Publishers Inc., San Francisco (2007)
4. Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection: A survey. ACM Comput. Surv. 41, 15:1–15:58 (2009)

5. Ciccio, C.D., 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 Transactions on Ambient Systems 11(10-12) (2011)
6. Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, pp. 226–231. AAAI Press (1996)
7. Frank, E., Witten, I.H.: Generating accurate rule sets without global optimization. In: Proceedings of the Fifteenth International Conference on Machine Learning, pp. 144–151. Morgan Kaufmann Publishers Inc., San Francisco (1998)
8. Heß, A., Johnston, E., Kushmerick, N.: ASSAM: A Tool for Semi-automatically Annotating Semantic Web Services. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 320–334. Springer, Heidelberg (2004)
9. ISO 29341-1:2008: Part 1: UPnP Device Architecture Version 1.0. International Organization for Standardization, Geneva, Switzerland
10. Kaldeli, E., Warriach, E.U., Bresser, J., Lazovik, A., Aiello, M.: Interoperation, composition and simulation of services at home. In: Eigth International Conference on Service Oriented Computing, pp. 167–181 (2010)
11. Khalid, B., Embury, S.M., Paton, N.W., Stevens, R., Goble, C.A.: Automatic annotation of web services based on workflow definitions. ACM Trans. Web 2, 11:1–11:34 (2008)
12. Lerman, K., Plangprasopchok, A., Knoblock, C.A.: Automatically labeling the inputs and outputs of web services. In: Proceedings of the 21st National Conference on Artificial Intelligence, vol. 2, pp. 1363–1368. AAAI Press (2006)
13. Li, F., Rasch, K., Truong, H.L., Ayani, R., Dustdar, S.: Proactive service discovery in pervasive environments. In: Proceedings of the 7th International Conference on Pervasive Services, pp. 126–133 (2010)
14. Li, F., Sehic, S., Dustdar, S.: Copal: An adaptive approach to context provisioning. In: 2010 IEEE 6th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), pp. 286–293 (2010)
15. Patil, A.A., Oundhakar, S.A., Sheth, A.P., Verma, K.: METEOR-S web service annotation framework. In: Proceedings of the 13th International Conference on World Wide Web, WWW 2004, pp. 553–562. ACM, New York (2004)
16. Rasch, K., Li, F., Sehic, S., Ayani, R., Dustdar, S.: Context-driven personalized service discovery in pervasive environments. World Wide Web 14(4), 295–319 (2011)
17. Wu, D., Parsia, B., Sirin, E., Hendler, J., Nau, D.: 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)