Serral Asensio, E.; Valderas Aranda, PJ.; Pelechano Ferragud, V. (2013). Context-Adaptive Coordination of Pervasive Services by Interpreting Models during Runtime. Computer Journal. 56(1):87-114. doi:10.1007/s10270-013-0371-3.

Context-Adaptive Coordination of Pervasive Services by Interpreting Models during Runtime

Estefanía Serral, Pedro Valderas, Vicente Pelechano

Centro de Investigación en Métodos de Producción de Software (ProS)

Universitat Politècnica de València

C/ Camí de Vera S/N, Valencia, 46022, Spain

{eserral, pvalderas, pele}@dsic.upv.es

ABSTRACT

One of the most important goals of pervasive systems is to help users in their daily life by automating their behaviour patterns. To achieve this, pervasive services must be dynamically coordinated, executed, and adapted to context according to user behaviour patterns. In this work, we propose a model-driven solution to meet this challenge. We propose a task model and a context ontology to design context-adaptive coordination of services at a high level of abstraction. This design facilitates the coordination analysis at design time and is also reused at runtime. We propose a software architecture that interprets the models at runtime in order to coordinate the service execution that is required to support user behaviour patterns. This coordination is done in a context-adaptive way and decoupled from service implementation. This approach makes the models the only representation of service coordination, which facilitates the maintenance and evolution of the executed service coordination after deployment.

Keywords: context-adaptivity, service coordination, model interpretation at runtime.

1. Introduction

Pervasive computing is a computer paradigm that promises to make life simpler via electronically enriched environments that sense context**,** adapt to it, and automatically act accordingly in order to meet user needs [1]. The base of a pervasive system is a composition of services that are in charge of managing the objects of its environment allowing us, for instance, to switch lights on, raise blinds or detect the presence of a person.

One of the most important goals of pervasive systems is to help users in their daily lives by automating their behaviour patterns. These patterns are defined as a set of tasks that users habitually repeat in similar contexts [2]. By automating a user behaviour pattern, we mean that the system automatically coordinates the execution of the services needed to support the pattern. An example of this type of service coordination is the following: At 7:50 a.m. on a working day, the system switches on the bathroom heating; ten minutes later, when the user must get up, the system prepares a bath and plays the user's preferred music; then, if it is a sunny day, the system raises the bedroom blinds; otherwise, the system switches on the bedroom light; afterwards, when the user enters the kitchen, the system makes a coffee.

To achieve this automation, pervasive services must be dynamically coordinated according to context, i.e., to aspects related to time (at 7:50 a.m. on a working day, when the user goes into the kitchen), user preferences (the user's preferred music), the environment state (if it is a sunny day), etc. Since context continuously changes at runtime, service coordination is usually supported by manually encoding complex decision logic that implements the system response according to specific context changes. This solution leads to a time-consuming and error-prone activity that makes the context-aware service coordination difficult to maintain and evolve after deployment. Thus, these systems need to be stopped, modified, recompiled, and redeployed so that they can be evolved when user needs change or even when a misunderstood requirement is detected.

To solve these problems, we need solutions that allow us to design context-aware coordination of services by using high level abstractions. This will allow us to determine at design time if the designed coordination properly supports the user behaviour patterns. This designed coordination should be decoupled from service implementation as much as possible in order to facilitate their further evolution. In addition, these designs must be used at runtime to dynamically drive the adaptation of the service coordination according to context. We call this service coordination as context-adaptive to emphasize that context is explicitly used so that the system adapts the service coordination at runtime[1] [3].

In this work, we confront these challenges from a model-driven perspective. We propose a context-adaptive task model to define complex behaviours by means of the coordination of pervasive services. This model has mechanisms to indicate how this coordination must adapt according to context conditions. An ontology-based context model is used to do this. This solution allows us to design the context-adaptive coordination of pervasive

1 The terms *context-aware* or *context-sensitive* are usually used in a more generic way for describing applications that use context; they do not necessarily have to imply an adaptation.

services that must be performed at runtime by using the task metaphor, which is conceptually close to user behaviour patterns and which facilitates the coordination analysis at design time.

In addition, we take design models a step further and reuse them during runtime [4, 5]. In this work, we show how using design models during runtime constitutes a valuable mechanism to coordinate context-adaptive pervasive services, making "the brain" of this coordination be totally decoupled from the service implementation. This facilitates the maintenance and evolution of the designed service coordination providing support for changing behaviour.

To achieve this, we propose a software architecture that can sense context, interpret the models, and execute the corresponding services at runtime. The software architecture has two main components: a context monitor, which is in charge of sensing context continually and updating the context model; and a model engine, which is in charge of interpreting the task model and executing the corresponding services when specific context conditions are fulfilled. It is worth noting that this software architecture is not coupled with the service implementation and can be used, like the models, by any system that implements pervasive services in Java/OSGi technology.

To summarize, the contributions of this work are the following:

- A novel approach for designing context-adaptive coordination of services in order to automate user behaviour patterns. This approach allows designers to describe service coordination in a technology-independent way using high-level abstraction concepts such as task, location, preference, etc.
- A software architecture that reuses design models during runtime to dynamically coordinate the execution of services in a context-adaptive way. It uses the proposed models as a decoupled runtime representation of the required coordination, which facilitates maintaining and evolving the service coordination when users' needs change. This is discussed in detail in Section 7.

The rest of the paper is organized as follows: Section 2 analyses the related work. Section 3 introduces some background that contextualizes and explains the motivation for this work. Section 4 introduces a technique to design context-adaptive coordination of pervasive services at a high level of abstraction. Section 5 presents a software architecture that interprets the designed models at runtime and automates the user behaviour patterns as specified. Section 6 shows some examples of the context-adaptive coordination of services that are achieved at runtime using our approach. Section 7 discusses the main benefits and drawbacks of the proposed work. Finally, Section 8 presents the conclusions.

2. Related Work

Related work can be grouped into rule-based approaches for context-adaptive service coordination, approaches for developing adaptive systems using models at runtime, and approaches that deal with service coordination using models at runtime.

Rule-based context-aware approaches focus on programming rules to execute service coordination when certain context conditions arise. Some relevant examples are the works presented by Henricksen et al. [6] and García-Herranz et al. [7]. Henricksen et al. [6] present a set of models to specify and support the development of context-aware approaches that provide reactive behaviour to context changes. To do this, the authors propose a graphical Context Modelling Language (CML) to specify context information. They also propose two programming models to program event-condition-action rules in order to support the automatic execution of services according to context conditions. García-Herranz et al. [7] present a solution to end-user programmable context-aware smart homes. They have designed a rule-based language in which context-dependent composite events can be codified in the form of Event Condition Action (ECA) rules with the possibility of using timers between actions. These rules are used to trigger the execution of services according to context.

These approaches have made great advances by taking context into account in the service execution. However, they hardwire the context-adaptive coordination of services into the functional system. Consequently, the system ability is limited to only coping with a fixed set of context changes that are included at system design time. In our approach, this coordination is specified by using high-level models which are directly interpreted at runtime. This improves the comprehension of the coordination and facilitates its evolution at runtime after system deployment.

Other approaches develop adaptive systems using models at runtime. Some relevant examples are the works presented by Morin et al. [8], Bencomo et al. [9] and Blumendorf et al. [10]. Morin et al. [8] propose a combination between model-driven and aspect-oriented techniques to support dynamic runtime reconfiguration. They dynamically compose aspects to produce a set of configuration models and then use these models to generate the scripts needed to adapt a running system from one runtime configuration to another. Bencomo et al. [9] propose the use of architectural models to support the generation and execution of adaptive systems. The authors represent the adaptation policy under the form of a state-transition system in which the states correspond to the system configurations and the transitions correspond to the adaptations between these configurations. Blumendorf et al. [10] focus on the development of adaptive user interfaces and their adaptation at runtime

combining multiple models. These models are self-contained to ensure executability. This approach allows developers to monitor, maintain, manipulate, and extend interactive applications at runtime and thus manage the continuously changing requirements of user interface development.

These works show the growing use of models during runtime for improving software characteristics; however, none of them focus on achieving service coordination that is adaptive to context.

In contrast to these works, several business process approaches make use of models at runtime with the purpose of dealing with context-adaptive coordination of services. Some outstanding examples are FollowMe [11] and (uWDL) [12]. FollowMe [11] is an infrastructure that combines an ontology-based context model, a workflow-based programming model and the OSGi framework. This infrastructure supports the development of pluggable context-aware applications described with pvPDL (Pervasive Process Definition Language), which is a workflow definition language that can use RDF queries to check context conditions. FollowMe can adapt to different domains by plugging corresponding domain contexts and context-aware applications. The Web service-based workflow language (uWDL) [12] describes service flows and provides the functionalities to select an appropriate service based on high-level contexts, profiles, and event information, which are obtained from various sources and structured by an ontology. uWDL uses Resource Description Framework (RDF) triplets (subject, verb, and object) of the ontology to express high-level context information as service transition conditions. This allows the approach to provide users with an adaptive service for their current situation.

Even though these approaches take advantage of using models at runtime, they can only use web services. In contrast, our approach provides a software architecture that is decoupled from service implementation. In addition, our approach specifies the coordination of services to automate user behaviour patterns using a task model. This model provides a notation that is closer to the concepts of user behaviour patterns than workflows, which are primarily intended for business world[2]. Thus, our approach helps to validate that user behaviour patterns are properly supported by the designed service coordination.

3. Background: Pervasive Services and Context

One of the goals of this work is to provide a technique that allows us to design how pervasive services must be coordinated in order to support user behaviour patterns. Note that these services must be previously available.

[2] The Business Rules Manifesto, http://www.businessrulesgroup.org/brmanifesto/BRManifesto.pdf

As a motivation example, we present a set of pervasive services that are in charge of interacting with different objects of a Smart Home. Rather than presenting technology implementations, we show the services from a conceptual point of view using a UML class diagram. As Figure shows, we consider pervasive services to be classes with operations that allow interaction with the physical environment. Note, however, that other strategies that consider pervasive services to be web services or distributed components could also be used.

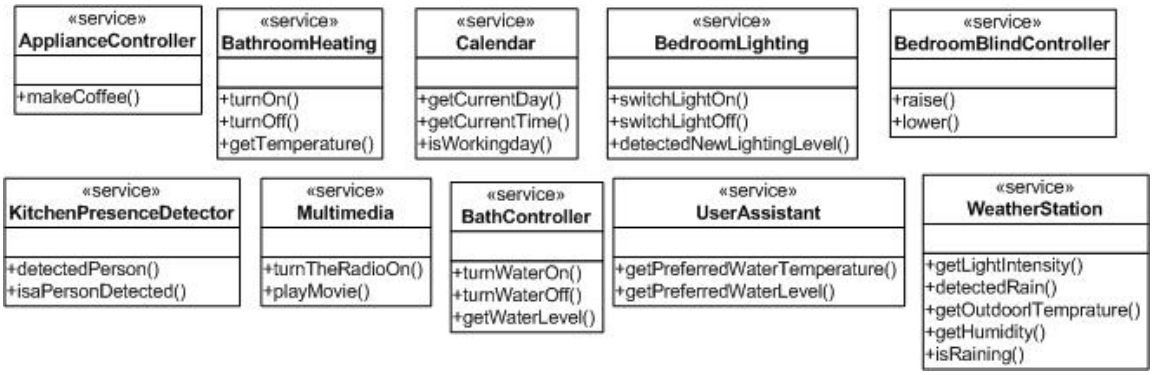

Figure 1. Example of pervasive services

The class operations presented in Figure 1 allow us to individually interact with different objects: we can turn the radio on and off; switch the light of the bedroom on and off; know if a person has been detected in the kitchen; etc. In this work, we focus on coordinating these individual interactions in order to provide the system with complex behaviours that satisfy user behaviour patterns.

This coordination must be dynamically adapted to context. There are different definitions of context within Pervasive Computing. However, the most commonly used definition is the one proposed by Dey in [13]. It defines context as any information that can be used to characterize the situation of an entity; an entity being a person, a place, or an object that is considered to be relevant to the interaction between a user and an application, including both the user and the application themselves.

Thus, service coordination must be adapted to properties that are related to people, places, objects, and the system; and these properties change over time (e.g. the location of a person is not the same throughout the day). In this sense, we want to remark that context changes are closely related to pervasive service execution. Note that services provide operations of two types: those that interact with actuator devices in order to provide the environment with an active behaviour (e.g. those that switch lights on, raise blinds, or activate an alarm); and those that interact with sensor devices in order to make the system aware of the context (e.g. those that sense the light intensity, detect the presence of a person, or check climatological conditions). Specifically, the interaction between services and actuator/sensor devices is performed as follows:

- **Services->Actuators**: the service operations that are in charge of controlling actuators are ***proactively*** executed by the system when it considers it opportune. Figure 1 shows some examples of these operations such as *BedroomLighting.switchOn(), BathHeating.turnOn(),* or *BedroomBlindController.raise()*. Note, however, that when these services are executed not only is an activator device controlled but a context change is also produced. For instance, each time the system switches the bedroom light on or off, a change in the environment is produced (the lighting intensity changes).
- **Sensors->Services**: the service operations that are in charge of interacting with sensors are ***reactively*** executed when sensors detect a context change. For instance, when a presence detector detects someone in the kitchen the *KitchenPresenceDetector.detectedPerson()* is reactively executed. These operations are combined with other ones that allow the system to check specific context conditions in a proactive way (e.g. *isaPersonDetected ()*);

Thus, context changes are produced in two ways (see Figure 2):

A) by proactively activating a service operation that interacts with an actuator. According to Dey's definition, the state of the system itself is part of the context. Thus, the execution of a service operation is considered to be a context change. In addition, the proactive execution of a service can change the environment state, which may trigger a reactive execution of another service.

B) by producing a change in the physical environment that activates a sensor, which at the same time produces the reactive execution of a service operation that informs the system about it.

Note how services are executed when context is changed in both cases (A and B). This constitutes a key pillar in sensing context within the software architecture that is presented in Section 5.

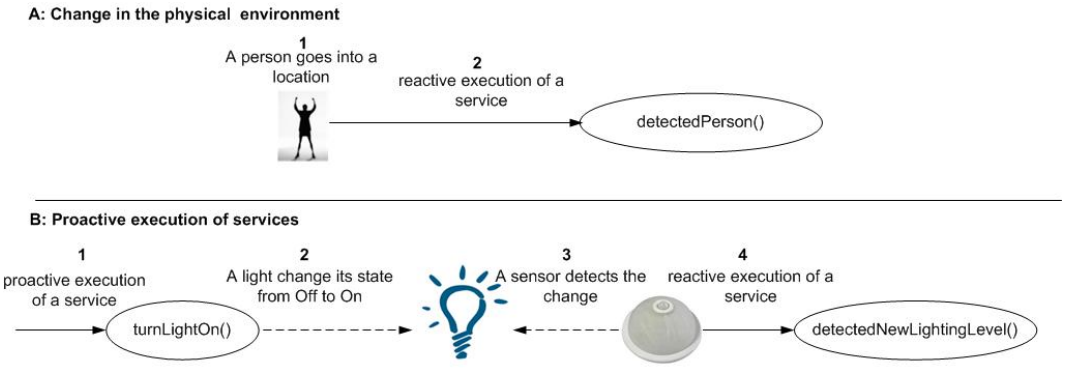

Figure 2. Changes in context

Thus, proactive and reactive service executions must be coordinated in order to properly support user behaviour patterns. Proactive executions are implemented by calling the corresponding services or creating the corresponding objects or components. Reactive executions are implemented by listener objects or similar entities that are created to "listen" at specific events. Both types of executions must be combined with traditional programming structures to specify conditions and iterations for the purpose of creating a coherent coordination.

```
when Calendar.getCurrentTime()==7:30 and Calendar.isWorkingDay():
    if BathroomHeating.getTemperature()<28 then BathroomHeating.turnOn();
    temp=UserAssistant.getPreferredWaterTemperature();
    wait(10);
    BathController.turnWaterOn(temp);
    when BathController.getWaterLevel()==UserAssistant.getPreferredWaterLevel():
  BathController.turnWaterOff();
    Multimedia.turnTheRadioOn();
    if not WeatherStation.isRaining() then BedroomBlindController.raise();
    else BedroomLighting.swithLightOn();
    when KitchenPresentDetector.isaPersonDetected():
         ApplianceController.makecoffee();
```

Figure 3. Example of runtime service coordination in pseudo-code

As a motivation example, Figure 3 presents a pseudo-code representation of the coordination that is required to satisfy the behaviour pattern that supports the awakening of a user (which has been presented in the introduction). Note that this coordination requires:

1. "Listening" at the reactive execution of the service operations that inform about context changes such as time and day changes, user location, etc. This is represented by conditions preceded by the `when` keyword.
2. The execution of service operations such as `WeatherStation.isRaining()` or `BathroomHeating.getTemperature()` that interact with sensors in order to proactively ask for the value of context properties. These properties are used in conditions preceded by the `if` keyword.
3. The proactive execution of service operations that interact with actuators such as `BathroomHeating.turnOn()` or `ApplianceController.makecoffee()`.

Let us assume that we have misunderstood the user's needs and that, after deployment, we have detected that this waking up behaviour pattern must be executed on weekends instead on working days; or even that the user's preferences have changed and s/he prefers an orange juice for breakfast instead of a coffee. Note that, with ad-hoc solutions such as the one presented above, these simple changes require stopping the system, modifying its source code and redeploying it. To prevent this, we require engineering solutions that allow us to design service coordination that can be easily validated before deployment. In order to facilitate post-deployment evolution, this solution should also allow us to implement the designed coordination decoupled as much as possible from the service implementation. These requirements lead to the research questions that are confronted in this paper, which are the following:

- How can the coordination of services be designed in order to easily validate the support for user behaviour patterns?
- How can service coordination be decoupled at runtime from the pervasive service implementation in order to facilitate its evolution?

Next, the answers to these questions are discussed in detail.

4. Designing Context-Adaptive User Behaviour Patterns

In order to answer the first research question presented above, we propose a design of context-adaptive coordination of services that is driven by user behaviour patterns. A behaviour pattern is a set of tasks that are repeatedly performed in similar contexts. Thus, two important aspects must be considered: 1) the context on which the behaviour pattern tasks depend; and 2) the tasks that must be performed by executing the pervasive services according to context.

To describe these aspects we propose the use of a context ontology and a context-adaptive task model.

4.1 A context ontology

Different context information is needed to coordinate services according to user behaviour patterns. In particular, we have identified the following information from previous works found in the literature [13-17]:

- Environment information: space information (areas of the environment where the system is running and spatial relations between these areas), environment properties (properties of the environment, e.g.: temperature, light intensity, etc.), etc.

- System information: information about the system, such as the services that it provides, the devices of the system, their computational resources, etc.
- User information: personal data, preferences, skills, location, mobility, etc.
- Privacy and security policies: information that indicates what actions each user can execute and what context information each user can see and modify.
- Temporal information: date and time, holiday, working day, etc.
- Event information: information about the events that happen in the system, such as user actions and context changes.

Different context models have been proposed to capture context in Pervasive Computing. Some of the most remarkable examples are: object-oriented models such as the ones proposed by the projects CORTEX [18] and Hydrogen [19]; directed state graphs such as the one proposed in [20]; key-value models such as the one used by Dey in the Context Toolkit [13]; graphical models such as ContextUML [21], CML [22] and the one proposed in [23]; etc.

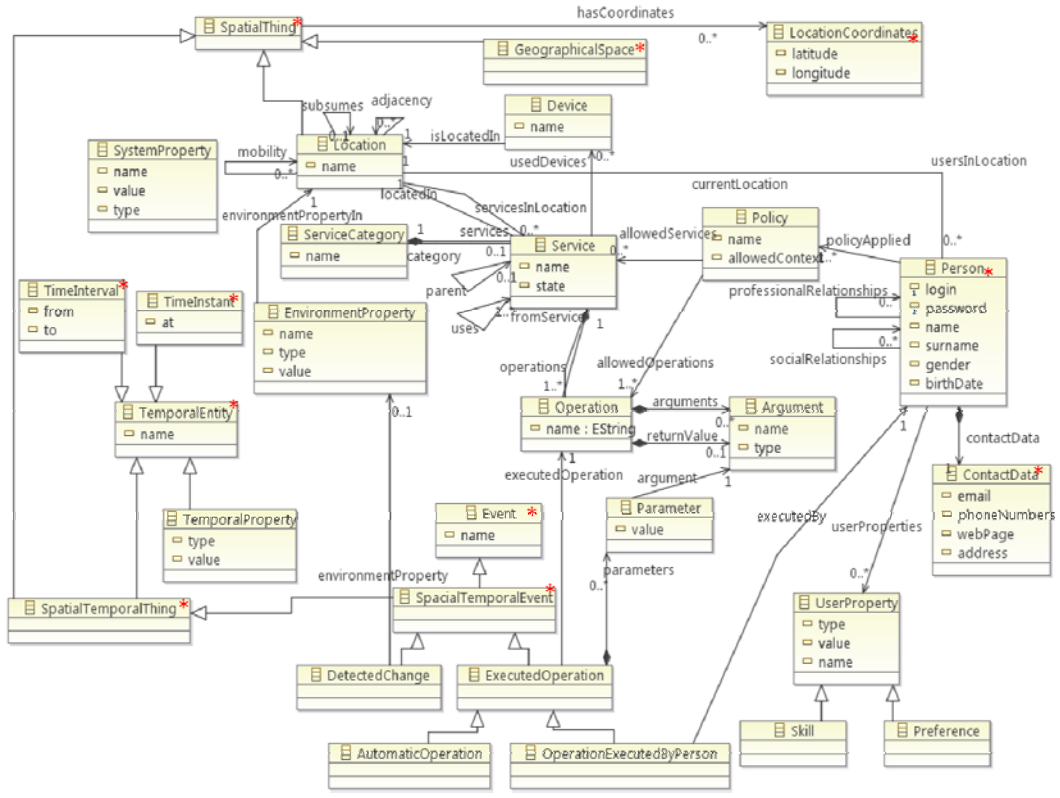

Figure 4. Class diagram of the context ontology

However, several studies [14, 24, 25] state that the use of ontologies to model context is one of the best choices. They state that this model guarantees a high degree of expressiveness, formality, and semantic richness. Ontologies also exhibit prominent advantages for reasoning and reusing context as well as facilitating the

integration of pervasive environments. Some relevant examples of ontology-based approaches are SOUPA [14], SOCAM [26], Ontonym [27] and COMANTO [28]. A complete background of the ontologies proposed in Pervasive Computing can be found in [25].

We base our context model on the SOUPA ontology. It is the most consistent set of ontologies since it imports most of its concepts from external and consensual domain ontologies [25]. We extend it to capture data related to aspects such as user preferences, skills, privacy and security policies, or user past actions in a more suitable way for our purposes. To build the context ontology, we have followed a top-down approach, starting from the most coarse-grained terms and breaking them down into finer-grained terms. The coarse-grained terms that we identify are: *Environment*, *System*, *Person*, *Policy*, *Time*, and *Event*. After breaking these terms into finer-grained ones, we obtain the classes of the class diagram shown in Figure 4 (where the concepts that are reused from SOUPA are marked with an asterisk). Table 1 links and compares the definition of these concepts with the ones defined in the above quoted examples of relevant context ontologies.

Persistence and Implementation. The concepts presented above constitute an abstract description of the context information that is required to support user behaviour patterns. However, this ontology must be interpreted at runtime to automate the behaviour patterns (which is explained in Section 5). Therefore, the concepts of the ontology must be implemented in a concrete technology that facilitates the interpretation of the on1tology at runtime. We have used the Web Ontology Language (OWL)[3], which is machine-interpretable and a W3C standard. In OWL, the classes of the ontology are directly defined by OWL classes. To do this, we can use existing OWL tools such as SWOOP[4] , Altova[5] or Protègè [6].

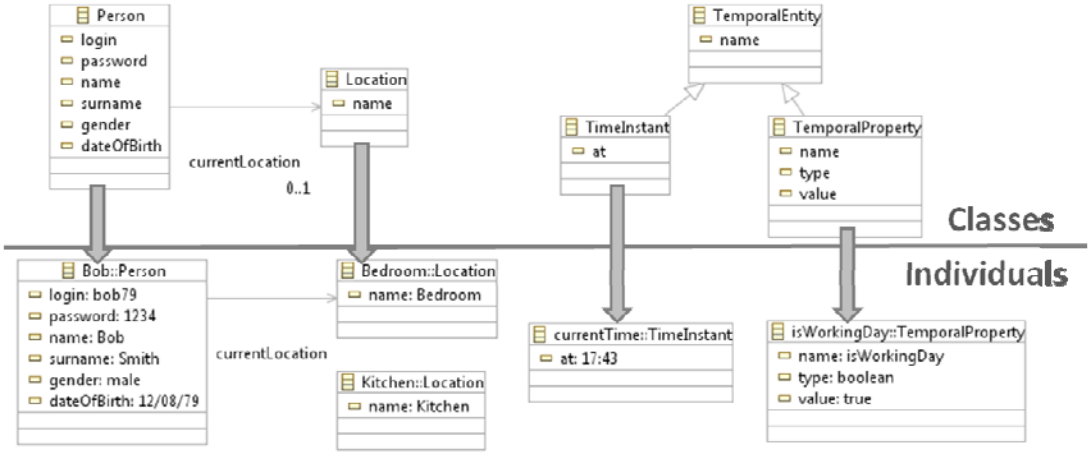

Figure 5. Instantiation of OWL Classes in Individuals

[3] http://www.w3.org/TR/owl-guide/
[4] http://www.mindswap.org/2004/SWOOP/
[5] http://www.altova.com/products/semanticworks/semantic_web_rdf_owl_editor.html
[6] http://protege.stanford.edu/overview/protege-owl.html

Main ontology concepts	Related Ontologies
Environment	To describe the environment, we reuse the OpenCyc Spatial and RCC **SOUPA** ontologies, which allow space to be specified using geo-spatial coordinates or symbolic geographical representation. We extend this representation with the term *EnvironmentProperty*, to allow any property of the environment to be defined (e.g., temperature, presence, etc.). We also add the term *Location* to describe in a more intuitive way the different areas that compose the environment (i.e., Kitchen, Corridor, etc.). This concept is similar to the *SymbolicRepresentation* concept that **Ontonym** proposes. Thus, Ontonym supports spatial descriptions as physical positions but also as symbolic positions. **SOCAM** provides the *Location* class that is represented by its coordinates. Each location also describes its temperature and its noise level. The Location class can be specialized in *IndoorSpace* and *OutdoorSpace*, and these terms must be, in turn, specialized to domain-specific locations such as room, kitchen, etc. In our ontology, these classes would be individuals of our Location class. **COMANTO** proposes the class *Place* that associates a physical location with its symbolic or geographic representation.
System	To describe the system, we propose the terms *Service*, *ServiceCategory*, *Operation*, *Argument*, *Device* and *SystemProperty*. **SOCAM** and Ontonym give support for device representation. **COMANTO** proposes the classes *Sensor* and *Service*. The *Sensor* class is like our *Device* class. Its *Service* class captures information relevant to the services/applications the user has subscribed to. The operations that these services provide are not represented.
Person	To describe the users of the system, we reuse the FOAF **SOUPA** ontology, which propose the term *Person*. To properly describe the users, we add the *UserProperty* class, to represent the properties of users, such as its preferences (e.g., preferred music, preferred language, etc), or its skills and disabilities (e.g., computer knowledge, blindness, etc.). With regard to the location in which a person is, we propose also the *currentLocation* relationship, which relates each person to the location where it is in the current moment. **Ontonym** provides terms for describing the identity, personal details and social relationships of people. **SOCAM** also proposes the *Person* class and use the FOAF ontology for defining people. **COMANTO** presents the concept *Person* for incorporating the user related context and representing person-to-person associations. It also provides the *Preference* class and distinguishes four preference subclasses namely: *DevicePreferences*, *ServicePreferences*, *NetworkPreferences* and *OtherPreferences*.
Policy	Our ontology allows us to describe the services of the system and the operations that they provide; therefore, we describe a policy as a set of operations and/or services (which group a set of operations) that are permitted for a person. In addition, a policy also describes the context information that a person can see and/or modify. **SOUPA** uses the Rei Policy Ontology. It specifies high-level but more complex rules for granting and revoking the access rights to and from different actions (concept similar to the *Operation* concept that our ontology provides, but focused only on agents).
Time	To describe temporal aspects, we reuse the DAML-Time ontology and the Entry Sub-ontology of Time that **SOUPA** provides. These ontologies provide us with the term *TemporalEntity*, which is refined into *TimeInstant* and *TimeInterval*. *TimeInstant* is defined by using the *at* property that stores the value of time; while TimeInterval is defined by using the *from* and *to* properties that relate the time interval to the two corresponding time instants. In addition, temporal relationships to compare and order to different temporal entities are also provided (e.g., *after,before, sameTimeAs, startsLaterThan, etc.).* For avoiding overloading the model, we do not show these relationships in Figure 4. To these classes, we added the *TemporalProperty* class as another refinement of the *TemporalEntity* class. It represents symbolic temporal properties that are not identified as a time instant or a time interval, such as the day of the week, if it is holidays or working days, etc. As SOUPA, **Ontonym** and **COMANTO** also support time instant and time interval representation.
Event	To describe the events that happen in the system, we reused the *Event* class proposed by **SOUPA**. In SOUPA, an event is a temporal and spatial thing. Thus, SOUPA provides the *SpatialTemporalThing* class, which is the intersection between *TemporalEntity* and *SpatialThing*. In addition, the *SpatialTemporalEvent* class is defined as the intersection of the *Event* and *SpatialTemporalThing* classes. Thus, in order to better represent the events of our systems, we refine the *SpatialTemporalEvent* class as shown in Figure 4 to differentiate if the event is a change detected by sensors or the execution of an operation executed by a person or automated by the system. **Ontonym** supports spatial temporal events like SOUPA, but Ontonym also allows an event to be related with an entity.

Table 1. Comparison of the main ontology concepts to related ontologies

Furthermore, the concepts of the ontology must be instantiated for a specific pervasive system and environment. This is done by specifying OWL *individuals*, which are instances of the OWL classes. For instance, the ontology for a pervasive system that controls the smart home of a user named Bob may have individuals such as *Bob* (which instantiates the class *Person*), *Kitchen* and *Bedroom* (which instantiate the class *Location*), or *currentTime* and *isWorkingDay* (which instantiate the classes *TimeInstant* and *TemporalProperty*, respectively). Figure 5 shows these individuals in a graphical way. Note that some of these individuals represent information that changes over time (such as the *currentTime* and *isWorkingDay* properties, or the *currentLocation* of Bob). These individuals are updated at runtime by the software architecture presented in Section 5.

4.2 A context-adaptive task model

Considering the behaviour modelling techniques proposed (such as Use Case Diagram, Task Model, Interaction Diagram, Business Process Model, etc.), we selected task models because they provide a notation intuitive and well understood by the users [29] to describe behaviour patterns. In addition, task models can provide enough expressivity to describe user behaviour [29] and to do it in a machine understandable way [30]. Thus, task models can very useful to specify behaviour patterns in such a way that they can be automated from their specification (by interpreting the model at runtime).

Task modelling is centred around users' own experiences and goals [31] and has proven to be effective in several areas such as user-interface modelling [32-34], assisting end-users in the execution of tasks through service provisioning and resource allocation [35, 36], etc. For this reason, numerous task model formalisms and methodologies have been developed (see [30] for task modelling comparison). Some of the most important ones are GOMS [30], TKS [37], HTA [38], CTT [32] and UsiXML [34]. These works show the growing use of task modelling and its remarkable results and possibilities to model user interaction with the system. However, they have not been proposed with the goal of automating this interaction (i.e., with the goal of automating user behaviour patterns). Therefore, these task models do not provide enough expressivity to specify whole behaviour patterns (such as specific relationships between tasks, context conditions, etc.) or to accurately describe behaviour patterns so that they can be directly executed.

Thus, we propose a model that provides a notation flexible enough to define both the different tasks that take part in a behaviour pattern and the context conditions that define when these tasks must be performed. We call this model a context-adaptive task model. This model is based on Hierarchical Task Analysis (HTA), which

has proven to be successful in a large number of projects. The basic idea of HTA is to describe the set of tasks in different abstraction levels as a task hierarchy. Figure 6 shows the meta-model that describes the elements that can appear in the task model. This meta-model includes terms such as *Task*, *Refinement*, and *TemporalRelationship*, which focus on defining the coordination of pervasive services; and other terms, such as *ContextProperty*, *ContextPrecondition* and *ContextCondition*, which focus on defining the context conditions in which services should be executed. Next, we explain how these elements are used to both coordinate pervasive services and adapt the coordination to context.

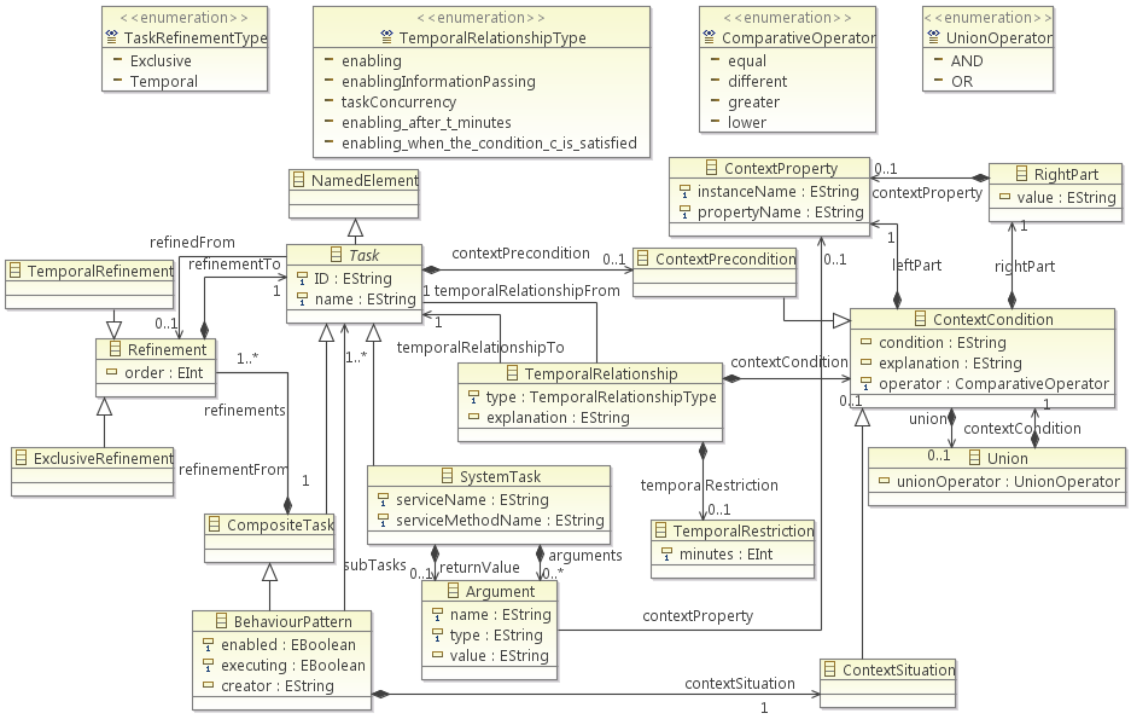

Figure 6. Task model metamodel

Coordination of Services. Using HTA, we describe each behaviour pattern as a task hierarchy. For instance, Figure 7 shows the *WakingUp* user behaviour pattern, which illustrates the tasks that the system must perform when a user is woken up. The root task represents the behaviour pattern, which is progressively broken down into simpler tasks by means of two task refinements: The exclusive refinement and the temporal refinement.

The exclusive refinement (represented by a solid line), which decomposes a task into a set of subtasks in such a way that only one subtask will be executed (disabling the others). In this type of coordination, the adaptation to context will play a main role (which is further explained below) since context will indicate which subtask must be executed.

The temporal refinement (represented by a dashed line), which also decomposes a task into a set of subtasks; however, all the subtasks must be performed in a specific order. These ordered tasks constitute what is known in HTA as a task plan (a coordinated execution of tasks). To indicate the task order, this refinement provides constraints that are depicted by means of arrows between the refined subtasks. There are two types of constraints:

- Those that represent specific task order restrictions, which are defined by means of the LOTOS (Language of Temporal Ordering Specification) operators [39]. For reasons of brevity, we only introduce those operators that are used in the example shown in Figure 7:
    - T1 >> T2: the T2 task is triggered when the T1 task finishes. For instance, in the *WakingUp* pattern defined in Figure 7, the system must adapt the bedroom after turning on the bathroom heating.
    - T1 [] >> T2: the T2 task is triggered when the T1 task finishes. T1 produces an output work product that is used as an input work product by T2. The work product is depicted between brackets below the arrow that represents the constraint. For instance, in the *WakingUp* pattern, the system must turn on the bath water after getting the preferred user water temperature. The temperature is the output work product of the first task (depicted as [temp]) and the input work product of the second task.
    - T1 ||| T2, concurrent tasks: T1 and T2 can be performed in any order or at same time. For instance, the system can prepare the bath, turn the radio on and manage lighting in any order.
- Those that represent context conditions, which are based on the context properties defined in the ontology presented in Section 4.1. This type of constraint is presented in detail below.

The refinement of tasks is finished when all the leaf tasks can be related to a service provided by the system. Note that a task represents a goal, while a service is the performance of this goal [40]. Thus, behaviour patterns should be described taking into account the services that are available in the system. If a leaf task cannot be supported by a pervasive service, it indicates that the behaviour pattern has not been defined in a realistic way (i.e., taking the services provided by the system into account). In this situation, either the behaviour patterns should be redefined or the pervasive system should be extended with new services.

According to the behaviour pattern shown in Figure 7 (tasks that represent services are depicted with a highlighted border) the waking of the user is supported as follows: the system turns the bathroom heating on, and

then, adapts the bedroom. After adapting the bedroom, the system makes a coffee. To adapt the bedroom, the system prepares a bath, turns the radio on, and manages bedroom lighting in a concurrent way. A bath is prepared by getting the user temperature, turning the bath water on, and turning it off later. Note that we do not indicate when to perform this last task; to indicate this, context conditions need to be used. To manage the room's lighting, the system can either raise the bedroom blinds or switch bedroom light on.

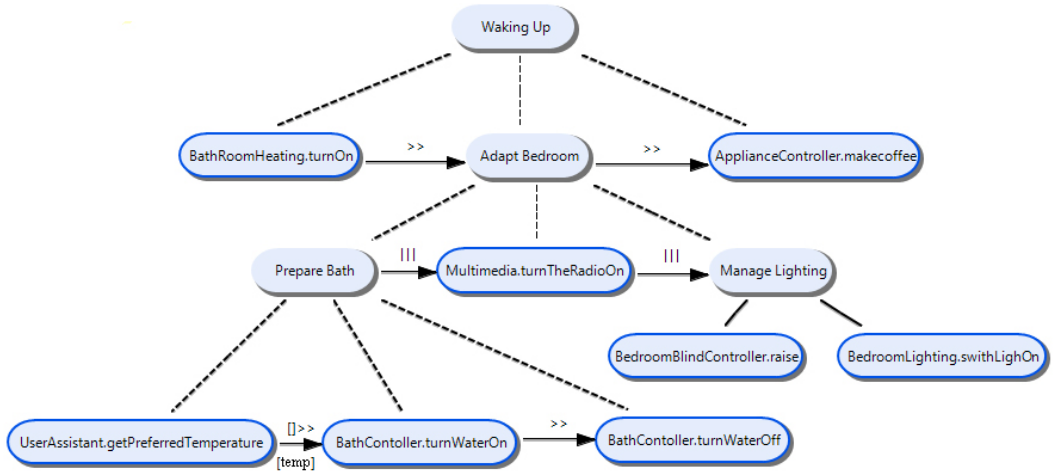

Figure 7. Example of behaviour pattern modelling

Context-Adaptation. To properly support the *WakingUp* user behaviour pattern, we need to consider specific context conditions in the coordination of the services shown above. For instance, the `BathfoomHeating.turnOn()` service operation should not be executed if the bathroom temperature is high. Thus, the execution of this operation must be adapted according to the bathroom temperature.

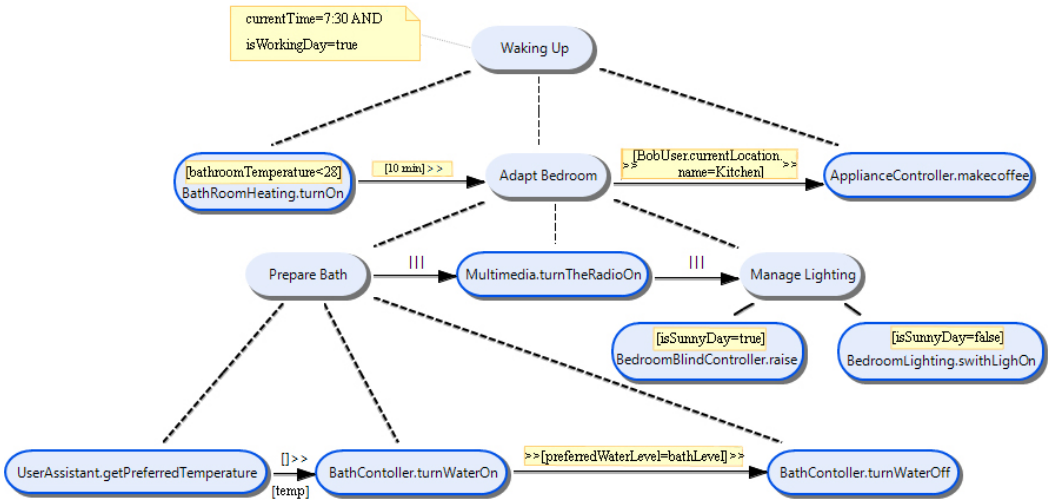

Figure 8. Example of a context-adaptive behaviour pattern

To achieve this context adaptation, we extend the task model with elements that allow us to define the context conditions that must be satisfied in order to execute services (see Figure 6). These conditions are evaluated as true or false and are composed of one or more simpler conditions linked together by the following logical connectives: and (AND), or (OR), equalities (=), inequalities (!=) and greater (>), or less than (<). The simplest condition is described as a logical expression composed of a left member (which is a property of the context model), a comparative operator, and a right part (which is a value or another property of the context model). Thus, context conditions are constructed in a similar way to the restrictions presented in PERCOMOM [41] for evaluating context. To refer to a context property, the name of the context property specified in the context model and the name of the individual to which this property belongs have to be indicated (e.g., a context property could be the *value* property of the *Temperature*). Using context conditions, we extend the task model with the following elements:

- **A context situation**, which is associated to a behaviour pattern. It indicates the set of context conditions that must be satisfied in order to execute the full set of coordinated services. It is depicted by a UML[7] note associated to the root task of the hierarchy that defines the behaviour pattern. For instance, Figure 8 presents a context-adaptive version of the *WakingUp* behaviour pattern. Its context situation indicates that the behaviour pattern should be executed every working day, at 7:30 a.m.

- **Task context preconditions**, which can be associated to any task of a hierarchy except the root task. They indicate that the task and its subtasks must be executed if and only if the task precondition is satisfied. They are defined over individuals of the context ontology and are depicted between brackets just before the task name. For instance, the system must *turn the bathroom heating on* only if the bathroom temperature is less than 28 Celsius degrees. These preconditions are essential for the tasks refined by using an exclusive refinement for indicating which subtask must be executed. The task that must be executed is the task whose precondition is fulfilled. For instance, to carry out the *Manage Lighting* task, the system must *raise bedroom blinds* when it is a sunny day, whereas the system must *switch bedroom light on* when it is not a sunny day.

- **Context-dependent constraints**, which can be defined to coordinate the execution of tasks that have been obtained from the temporal refinement of the same parent task. As explained above,

[7] UML, http://www.uml.org

constraints of this type are depicted by means of arrows between subtasks. These constraints are the following:

- T1 >> [c] >> T2: after executing T1, T2 is performed when the condition c is fulfilled. For instance, the system must m*ake coffee* after *adapting the room*, but only when the Bob user is detected in the kitchen (*BobUser.currentLocation.name=Kitchen*). In the same way, the task that *turns the bath water off* is performed after *turning it on*, but only when the bath water level has reached the user level preferred by the user.

- T1 t >> T2: after executing T1, T2 is performed when the period of time t has elapsed. For instance, 10 minutes after *turning on the bathroom heating* the system must *adapt the bedroom*.

Finally, note how this notation for designing service coordination has several advantages:

(1) The hierarchical representation of tasks provides a flexible mechanism to make them conditional on specific context constraints: the execution of a task (a hierarchy node), the execution of a set of coordinated tasks (a branch of the hierarchy), or the execution of the complete behaviour pattern (the full hierarchy).

(2) The use of abstract concepts (close to the user knowledge related to the required behaviour patterns) allows designers to focus on the tasks that must be performed, deferring technological decisions and implementation issues. This also facilitates validating the service coordination at design time.

(3) These designs are totally decoupled from service implementation since we only need to know a service identifier (in this example, names of classes and operations) and the context properties that must be considered.

(4) The notation provides us with enough expressiveness to accurately describe context-adaptive service coordination in such a way that they can be directly executed.

In addition, note that these advantages are of special interest if we consider that the proposed designs do not need to be translated into any kind of implementation. Instead, they constitute the service coordination implementation since they are directly interpreted at runtime.

Persistence and Implementation. The XMI (XML Metadata Interchange) language has been used to make the context-adaptive task model persistent in a runtime processable technology. To facilitate the creation of the

model, we have developed a graphical editor using the EMF and GMF plugins of the Eclipse platform[8]. Figure 9 shows a snapshot of this tool.

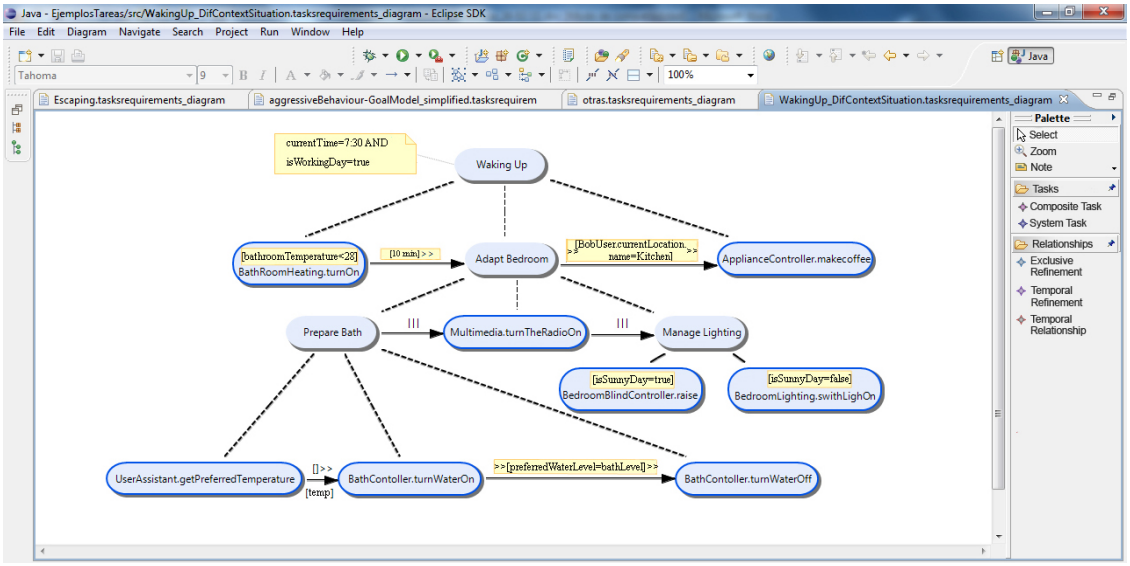

Figure 9. Snapshot of the graphical task-modelling tool

5. Executing Context-Adaptive Task Models at Runtime

The second research question stated in Section 3 requires a solution for implementing the coordination of context-adaptive services that is decoupled from the implementation of these services. This pursues the goal of facilitating the further evolution of the implemented coordination. To reach this goal, we propose the use of the design models presented above as the coordination implementation itself at runtime.

Using models during runtime [4] is an emerging development paradigm that proposes using models as runtime artefacts of first-order. It has been recently shown that models can provide a richer semantic base for runtime decision-making related to system adaptation and other runtime concerns. However, software infrastructures capable of interpreting the models at runtime are needed to achieve these benefits.

Therefore, we present a software architecture that interprets the design models at runtime and executes the pervasive services as designed. To develop the architecture, we follow a development process that is inspired by the work presented in [42]. This process has three main iterative activities: 1) design the architecture in a technology-independent way; 2) map the architecture to a specific technology; and 3) validate the technology-dependent architecture.

[8] Eclipse Platform, http://www.eclipse.org

5.1. Design of a Technology-Independent Architecture

In this section, we introduce a technology-independent definition of the proposed software architecture. The main goal of the proposed architecture is to execute the models presented above by interpreting them and executing the services required to support the user behaviour patterns. To execute the models, two important things have to be considered:

1. The opportune **context** in which each behaviour pattern has to be executed. The context that determines when behaviour patterns have to be executed is defined in the task model as context constraints (in context situations, preconditions, etc.); however, the values of the context properties used in those conditions are stored in the context ontology and must be continuously updated according to context changes (e.g., users change their position, lights are switched on/off, etc.).
2. The **tasks** to be executed in each behaviour pattern and how they must be executed. These aspects are defined in the task model.

Thus, the pervasive system architecture shown in Figure 10 has been defined. A *Context Monitor* and a *Behaviour Pattern Engine* play major roles in this architecture. Both the Context Monitor and the Behaviour Pattern Engine interact with the pervasive services and the runtime representation of the models presented in the previous section.

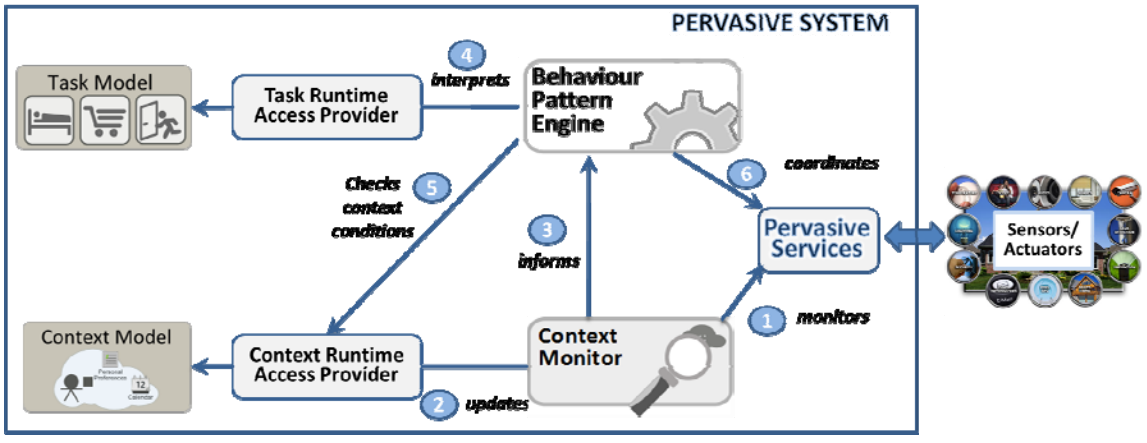

Figure 10. Technology-Independent Architecture

In a nutshell, the **Context Monitor** is in charge of monitoring the *execution of services* in order to capture context changes and update the context ontology accordingly at runtime. Note that we detect context changes by monitoring the service execution because of the way in which context is changed in pervasive systems (see Section 3). When a change in context is detected, the context monitor must also inform the *Behaviour Pattern*

Engine about it. The **Behaviour Pattern Engine** is in charge of 1) checking the context situations specified for each behaviour pattern, 2) selecting the leaf tasks, and 3) executing the services associated to those tasks, by taking into account the current context, the task relationships, and the task refinements. In order to interact with models at runtime, both the Context Monitor and the Behaviour Pattern Engine need to interact with the corresponding *Runtime Access Provider*. **Runtime Access Providers** are in charge of facilitating the manipulation of the task model and the context ontology at runtime. Finally, note that Services are in charge of interacting with sensor and actuator devices as explained in Section 3.

In order to better understand how the software architecture must behave, Figures 11 and 12 present two sequence diagrams. They illustrate the interaction between the different architectural elements to coordinate the execution of services at runtime.

5.1.1. Selection of the behaviour patterns to be executed

According to Figure 11, the selection of the behaviour patterns that need to be executed is performed as follows:

1. When a service is executed, the Context Monitor updates the ontology through the Context Runtime Access Provider.
2. Afterwards, the Context Monitor informs the Behaviour Pattern Engine about the context change.
3. Then, the Behaviour Pattern Engine asks for the set of behaviour patterns to the Task Runtime Access Provider, which retrieves them from the task model.
4. Next, the Behaviour Pattern Engine interacts with the Context Runtime Access Provider in order to check the behaviour pattern context situations that depend on the context change.
5. For each behaviour pattern whose context situation is fulfilled, the Behaviour Pattern Engine executes the task sequence described in the sequence diagram *executeBehaviourPattern()* shown in Figure 12. This sequence diagram explains how the Behaviour Pattern Engine must execute the coordination of services.

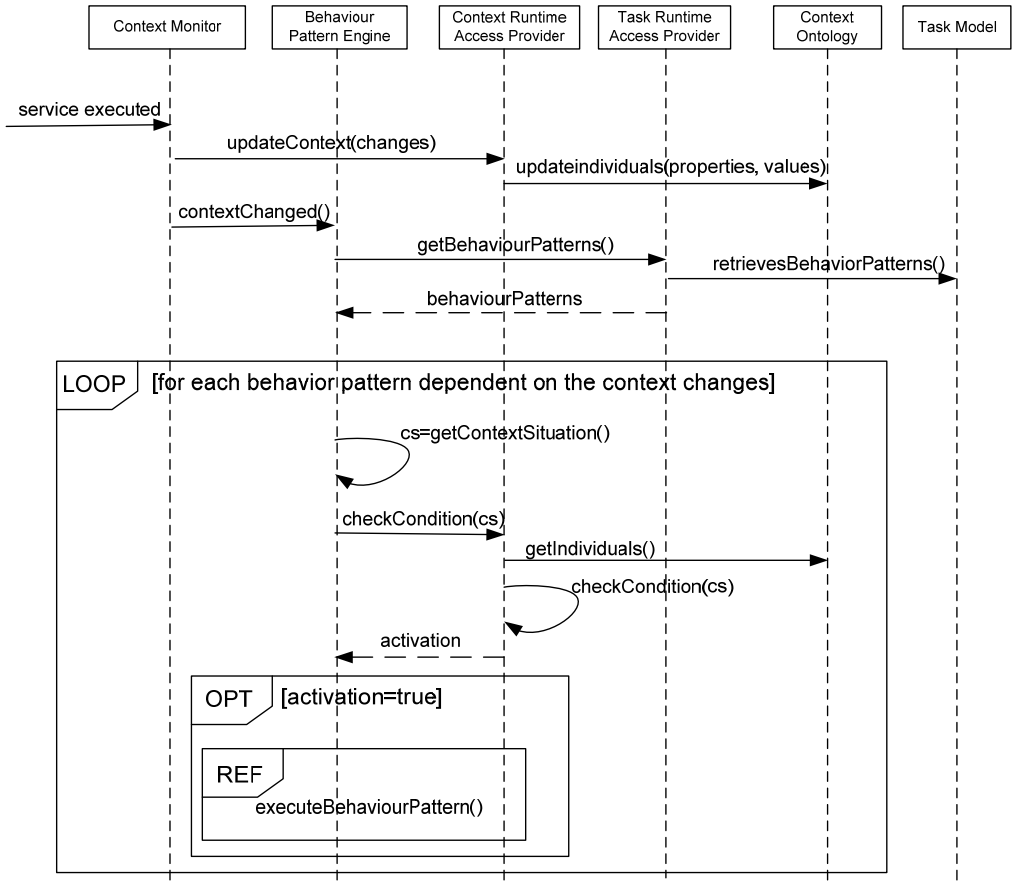

Figure 11. Coordination of Services at Runtime

5.1.2. Execution of behaviour patterns

According to Figure 12, each behaviour pattern is executed as follows:

1. The Behaviour Pattern Engine progressively retrieves the different leaf tasks (`bp.nextTask()`) included in the behaviour pattern (represented by the `bp` reference). In order to retrieve a next task, if it is refined by an exclusive refinement, the order of specification must be considered; if the task is refined by a temporal refinement, the temporal relationship defined in the behaviour pattern must be considered. In this last case, the next task is not retrieved until the context constraint associated to the temporal relationship that precedes the task is fulfilled. Note the `waitforRelContextContraintFulfillement()` call, which interacts with the Context Runtime Access Provider. This is a blocking call: the sequence diagram does not continue until the call is satisfied. For instance, for the behaviour pattern described in Figure 8, the task *turn the bath water off* is not retrieved until the bath level is at the level preferred by the user. If the next task is refined by an exclusive refinement, or it has no predecessor relationship, or the temporal relationship has no context constraint, then *true* is used as the constraint.

2. After retrieving a task, the Behaviour Pattern Engine interacts with the Context Runtime Access Provider to check the context precondition that may be associated to the task. See, for instance, the tasks that *turn the bathroom heating on*, r*aise bedroom blinds*, and *switch bedroom light on* in Figure 8. Note that not all the tasks are associated to a context precondition. If a task does not have a context precondition, then *true* is used as the precondition.
3. Finally, if the context precondition is fulfilled, the Behaviour Pattern Engine executes the corresponding service and performs all these steps for the next task.

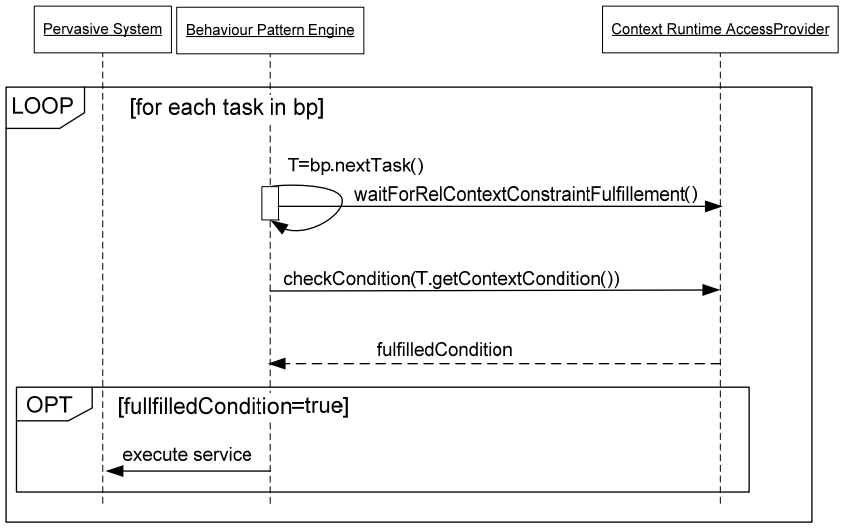

Figure 12. Execution of a Behaviour Pattern

5.1.3. Managing models

The Runtime Access Providers are in charge of providing access to the task model and the context ontology at runtime. To do this, they must provide constructors that facilitate the selection, query, and update of the different elements that can appear in the context ontology and the task model.

The task model is accessed by the Behaviour Pattern Engine in order to retrieve user behaviour patterns as well as other model elements such as tasks, relationships, or contexts conditions. Thus, mechanisms to query the task model must be provided by the Task Runtime Access Providers. In the sequence diagrams presented above, we have represented these mechanisms by object-oriented calls such as *getBehaviorPatterns()*, *nextTask()*, or *getContextCondition()*.

The context ontology is updated by the context monitor when a context change is produced. The ontology is also queried by the Behaviour Pattern Engine in order to determine the way in which services must be executed. Thus, mechanisms to update and query the context ontology must be provided by the Context Runtime Access

Providers. In this sense, it is important to determine the context information that can be checked by the Behaviour Pattern Engine and the context information that must be updated by the Context Monitor.

The Behaviour Pattern Engine can check all the information stored in the context ontology. Note that the context conditions that are defined in the task model can be based on temporal information, environment properties, system state, or user preferences. Thus, the Behaviour Pattern Engine must be able to check every individual of the context ontology.

The information that must be updated by the Context Monitor is related to the execution of services (both the proactive and the reactive execution, see Section 3):

- If services are proactively executed, the context monitor must store the exact operation that has been performed and the execution time. To store this information in the context ontology, the context monitor must create individuals of the *ExecutedOperation* class (see Figure 3), which constitutes a *TemporalEvent* (see the inheritance relationship). This class indicates a *TimeInstant* (note the *ExecutedOperation* class also inherits from *TimeInstant*) and a reference to the service operations (see the *executeOperation* reference).
- If services are reactively executed, the context monitor must store the environment properties that are modified and the modification time. To store this information in the context ontology, the context monitor must update the *EnvironmentProperty* individuals that represent the environment properties that have been modified. Also, the context monitor must create an individual of the *DetectedChange* class that indicates the *TimeInstant* and the *EnvironmentProperty* that has been changed.

As a representative example, Figure 13 shows graphically the individuals that must be created or updated by the context monitor when the system switches a light on in a proactive way, and when the system is informed about a change in the lighting level through the reactive execution of a service.

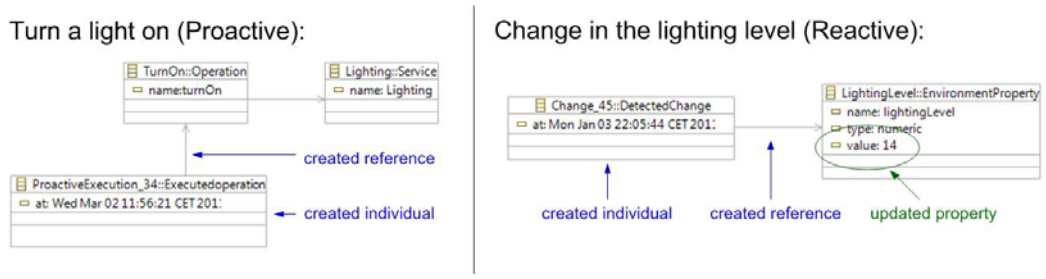

Figure 13. Example of context ontology updates

It is worth noting that there are other context properties that must also be updated, such as those related to time, system properties, people, or preferences. In order to manage properties related to time and system, the context monitor does not need to supervise the execution of services. To do this, it only needs to request the system for this information. For the properties related to people and preferences, the context monitor does not have to manage them. This context data should be updated by the users themselves. Section 7 explains how this is supported.

5.2 Mapping to a Specific Technology

In order to implement the architecture presented above, we have used the Java/OSGi technology. OSGi is the acronym for the Open Services Gateway Initiative, which provides a framework to implement service-oriented systems. We have used this technology because it is increasingly used for developing pervasive systems due to the important facilities that it provides for developing service-oriented systems. For instance, it allows services to be registered by publishing their interfaces in the framework's service registry. This registration allows a certain service to be searched for when needed. The framework also manages dependencies among services to facilitate their coordination among them. These dependencies are implemented by using Wire objects. A Wire object acts like a communication channel between a Producer service and a Consumer service. When a wire is created, the producer service can produce information to be used by the Consumer service. Services are marked as producers or consumers by using Java interfaces. Thus, when a Producer service is executed, OSGi automatically sends the produced data to its registered consumers so that they can analyze it.

The OSGi-dependent architecture is defined in Figure 14 in which Figure 10 has been annotated to show the used technologies. Next, we explain how the different elements of the architecture have been implemented as well as how the interaction defined in the sequence diagrams shown in Figures 11 and 12 is supported.

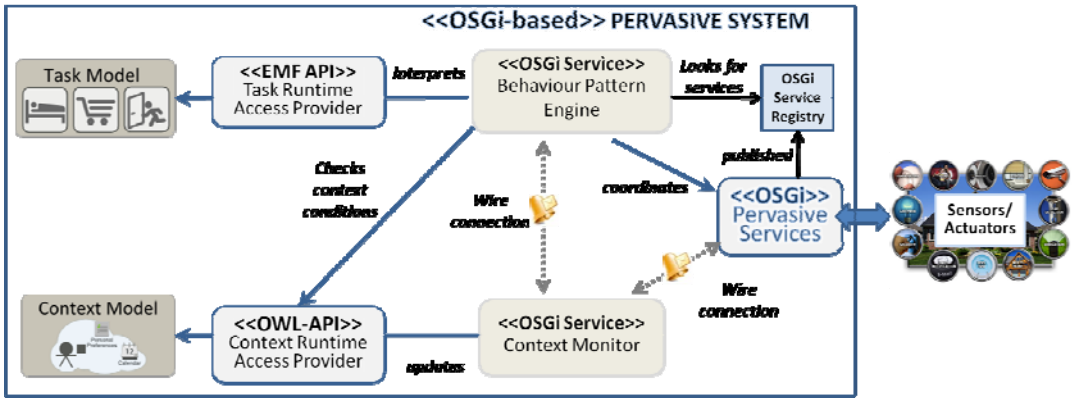

Figure 14. OSGi-dependent architecture

5.2.1. Selection of the behaviour patterns to be executed

To support the selection of the behaviour patterns that need to be executed, the following implementation has been done:

- According to the first steps of the sequence diagram in Figure 11, the Context Monitor must be aware of the execution of each pervasive service in order to detect context changes, and the Behaviour Pattern Engine must inform about these changes. To implement these, we have done the following:
    - The Context Monitor is developed as a Java OSGI service and the existing pervasive services are connected to it using a wire. In these wires, the services act as producers providing the context monitor with information about their execution, while the context monitor acts as a consumer because it uses the information produced by the services. Thus, the Context Monitor is aware of the execution pervasive services by means of Wire objects (see Figure 14).
    - The Behaviour Pattern Engine is also developed as a Java OSGI service. It is connected to the Context Monitor by an OSGi wire. In this wire, the Context Monitor plays the role of producer because it informs the interpreter about context changes, while the Behaviour Pattern Engine plays the role of consumer, because it needs to be informed by the Context Monitor.
- Besides informing the Behaviour Pattern Engine, the Context Monitor is also in charge of updating the context ontology. To do this, the OSGi service that implements the monitor makes use of the OWL facilities provided by the Context Runtime Access Provider (explained in Section 5.2.3).
- The next steps of the sequence diagram of Figure 11 show how the Behaviour Pattern Engine obtains the existing behaviour patterns and checks their context situation. To obtain the user behaviour patterns from the task model, the OSGi service that implements the Behaviour Pattern Engine makes use of the EMF facilities provided by the Task Runtime Access Provider. These facilities are introduced below. To check the context conditions, the OWL facilities provided by the Context Runtime Access Provider are used.

5.2.2. Execution of behaviour patterns

According to the first steps of the sequence diagram of Figure 12, the Behaviour Pattern Engine accesses the tasks that belong to a behaviour pattern. To do this, the different nodes of the task hierarchy must be accessed. To do this, we have followed the strategy used in [43] where leaf tasks are retrieved from hierarchical descriptions. Once a task has been retrieved and its context precondition has been checked, the corresponding pervasive service must be executed. The facilities provided by both Runtime Access Providers are used in order

to retrieve tasks and check conditions. The OSGi capabilities are used in order to execute the corresponding services. Since the services are published in the OSGi registry, the Behaviour Pattern Engine can search for (by using their identifier), retrieve and execute them (see Figure 14).

5.2.3. Managing models

The context ontology is accessed by the Context Runtime Access Provider, which is used by the Context Monitor and the Behaviour Pattern Engine to update and query the ontology, respectively (see Figures 11 and 12). To implement the Context Runtime Access Provider, we have used the OWL API, which is an open-source API that provides facilities for creating, examining and modifying an OWL ontology (note that the proposed ontology is implemented with OWL so that it can be used at runtime, see Section 4.1). To perform specific queries against the context ontology, we have used SPARQL[9] through the Pellet reasoner [44].

Thus, the Context Runtime Access Provider provides methods that facilitate the manipulation of the context ontology at runtime. As a proof of concept, Figure 15

presents the `setAttribute()` method that is used by the Context Monitor. This method updates the value of a property of an individual stored in the context ontology. This Runtime Access Provider provides other methods such as `setIndividual()` and `updateInidividual()`, which allow an individual to be created or updated, respectively; or `getIndividuals()`, `getAttribute()`, and `checkCondition()`, which allow the ontology state to be queried.

```
void setAttribute (String individualID, String propertyID, String newValue) {

       OWLIndividual individual=
                factory.getOWLIndividual(URI.create(prefixURI+individualID));
       OWLDataProperty dataProperty=
                  factory.getOWLDataProperty(URI.create(prefixURI+ propertyID));
       OWLDataType type= factory.getOWLDataType(URI.create(
                                "http://www.w3.org/2001/XMLSchema#String"));
       OWLConstant value=factory.getOWLTypedConstant(newValue, type);
       OWLDataPropertyAssertionAxiom assertion_data= factory.
           getOWLDataPropertyAssertionAxiom(individual, dataProperty, value);
       AddAxiom addAxiomData = new AddAxiom(ontology, assertion_data);
       manager.applyChange(addAxiomData);}
```

Figure 15. Changing the context property in the context model

The task model is accessed by the Task Runtime Access Provider, which is used by the Behaviour Pattern Engine to get the behaviour patterns (see Figure 10). To implement the Task Runtime Access Provider, we have

[9] SPARQL, http://www.w3.org/TR/rdf-sparql-query

used the EMF Model Query framework. This framework allows us to construct and execute query statements to retrieve model elements. Thus, the Task Runtime Access Provider provides the Behaviour Pattern Engine with the `getBehaviourPatterns()` method, which retrieves the list of patterns from the model, and other methods such as `getContextSituation()`, `getTasks()`, `getExecutionPlan()`, and `getRelationships()`, which facilitate the manipulation of the elements that belong to a specific behaviour pattern. As a proof of concept, Figure 16 presents the `getContextSituation()` method. The other methods are implemented in an analogous way.

```
boolean Contextsituation getContextSituation (String condition){
    SELECT query = new SELECT( new FROM(resourceModel.getContents()),
    new WHERE (new EObjectTypeRelationCondition(
       BehaviourPatternModelPachage.eINSTANCE.getContextSituation_Condition(),
                  new StringValue(condition))));
    IQueryResult res =  statement.execute();
    Contextsituation contextSituation = res.toArray()[0];
    return contextSituation;}
```

Figure 16. Example of a Task Runtime Access Provider operation

5.3 Validation of the technology-dependent architecture

We developed several validations to test the feasibility of the proposed architecture. Specifically, we have validated two aspects: (1) the correctness and completeness of the service coordination achieved through the execution of models, and (2) the scalability of accessing models at runtime. To make these validations, we used computers with the following features: Pentium 4, 3.0 GHz processor and 2 GB RAM with Windows XP Professional Edition SP3 and Java 1.5 installed. In addition, we used the implementation of OSGi *Prosyst Embedded Server 5.2*[10].

5.3.1. Completeness and Correctness

To evaluate the completeness of the architecture, we need to determine whether the Behaviour Pattern Engine executes all the services required by a behaviour pattern. We must also determine whether all the behaviour patterns that must be triggered when a context situation is fulfilled are triggered. In order to evaluate the correctness of the approach we need to determine whether the services required by a behaviour pattern are all

[10] Prosyst, http://www.prosyst.com

executed in the correct order and in the correct conditions. In the same way, we need to determine whether behaviour patterns are executed if and only if the proper context situation is fulfilled.

Since the Context Monitor registers each execution of a service in the ontology (see Section 6.1), the proposed validation consist in: (1) simulating the fulfilment of specific context conditions in order to trigger the execution of several behaviour patterns, and (2) checking that the context monitor properly registers the execution of all the services that must be executed (completion) and in the correct order (correctness). Therefore, we must previously validate other aspects: that the Runtime Access Providers properly retrieves and saves data, and that the Context Monitor properly registers service execution.

We used JUnit[11] tests to perform all these evaluations. We developed a set of JUnit that allow us to evaluate the proper behaviour of the Runtime Access Provider, the correct behaviour of the Context Monitor, and the completeness and correctness of the Behaviour Pattern Interpreter. As a representative example, Figure 17 shows the JUnit method that evaluates the completeness of a behaviour pattern execution. To perform this evaluation, we obtain the execution plan derived from the leaf tasks of the behaviour pattern according to the current state of the ontology. To do this, we use the `getExecutionPlan()` method that constitutes one of the EMF facilities provided by the Task Runtime Access Provider. Next, we execute the behaviour pattern through the method `executeBehaviourPattern(bp)`. Afterwards, we retrieve the last registered automated operations from the ontology (i.e., we retrieve the individuals of the *AutomaticOperation* class) by interacting with the Context Runtime Access Provider. We retrieve as many automated operations as tasks contained in the previous plan. Finally, we create an equal assertion to check whether or not the automated operations retrieved from the ontology are the same as the executed tasks that the plan contains.

```
public void executeBehaviourPattern(BehaviourPattern bp){
  List<Task> executionPlan=bp.getExecutionPlan();
  executeBehaviourPattern(bp);
  List<Operation> automatedOperations=ContextProvider.getLastAutomaticOperation
                                                    (executionPlan.size());
  ArrayList<String> plannedTasks, executedTasks;
  for(Task t: executionPlan) planedTasks.add(t.getName());
  for(Operation o: automatedOperations) executedTasks.add(o.getName());
  assertEquals(plannedTasks, executedTasks);
}
```

Figure 17. JUnit test example

[11] JUnit, http://www.junit.org/

We performed these evaluations in an iterative way, which allowed us to detect and resolve some mistakes. For instance, we realized that the behaviour patterns that are dependent on time, made the system enter into a loop. This was because the system updates time every second and the smallest time unit considered in the behaviour patterns was minutes. Consequently, the context situation of these patterns was continuously fulfilled until a minute passed. To solve this problem, we needed to use the same time unit in both cases. Since updating the context model every second could overload the system, we updated the context monitor so that the time was updated every minute.

5.3.2. Scalability

Models are manipulated at runtime by the Runtime Access Providers (see Section 6.1). This manipulation is subject to the same efficiency requirements as the rest of the system because the execution of model operations impacts the overall system performance. Therefore, these operations have to be efficient enough so that the system response is not drastically affected. In order to validate whether or not our approach scales to large systems, we quantified the temporal cost of the operation done with randomly generated large models.

To test these operations, we used the properties defined in the context ontology presented in Section 4.1 and an empty task model to be randomly populated by means of an iterative process. The context ontology was populated with 100 new context values each iteration, while the task model was populated with one new pattern whose task structure formed a perfect binary tree, varying its depth and the width of its first level each iteration.

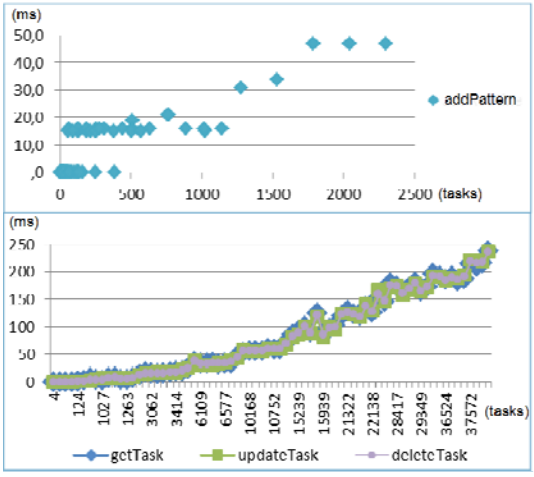

Figure 18 Temporal cost of accessing the task model at runtime

After each iteration, we tested all the model operations of the Runtime Access Providers 20 times and calculated the average temporal cost of each one. As an example, the operation of the Context Runtime Access

Provider with the highest temporal cost was the operation to check a context condition, which took 7 milliseconds with 100 individuals and 10 milliseconds with 6000 individuals. This is because this operation has to obtain the individual by using a SPARQL query, which determines the temporal cost of the operation. Figure 18 shows the temporal cost of the operations of the Task Runtime Access Provider with the highest cost. At the top of the figure, we show the time required to add a behaviour pattern according to the number of tasks. This operation took less than 50 milliseconds to add a pattern of 2296 tasks. At the bottom, we show the temporal cost of the operations for getting, updating, and deleting a task. These costs were very similar since all of them make the same query to obtain the corresponding task. Even with a model population of 45612 tasks, these model operations provided a fast response (250 milliseconds). These results show that the response time is not drastically affected as the size of the models grows.

6. Example Scenarios

In order to better understand how the proposed architecture works, we present some example scenarios. In these scenarios, we show the service coordination performed by the architecture at runtime in specific context situations. More examples and videos about this coordination can be found at http://www.pros.upv.es/art.

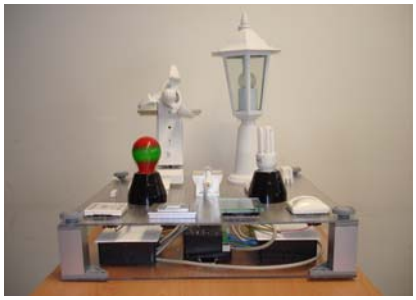

Figure 19 The device infrastructure with simulation purposes

To put these scenarios into practice, we edited the context ontology to set the context properties involved in the *WakingUp* user behaviour pattern (see Figure 6). Next, in order to simulate a real interaction with sensors, we used a device infrastructure (see Figure 19) specifically build with simulation purposes. This infrastructure contains several devices such as presence detectors, light intensity sensors, weather stations, blind controllers, and so on. To simulate the interaction with devices that are difficult to have in an academic environment (e.g., a smart bath), we have used a device simulator. This simulator[12] allows developers to define virtual devices and control them using an intuitive user interface.

[12] http://oomethod.dsic.upv.es/labs/projects/pervml

6.1 Coordination performed on a sunny day

A partial view of the state of the context ontology is shown in Figure 20. Summarizing: Bob is in the bedroom; it is 7:29 a.m. on a sunny working day; the bathroom temperature is 28 degrees Celsius; and Bob likes having a bath with the water at 35 degrees and the bath filled to 80 percent of its capacity.

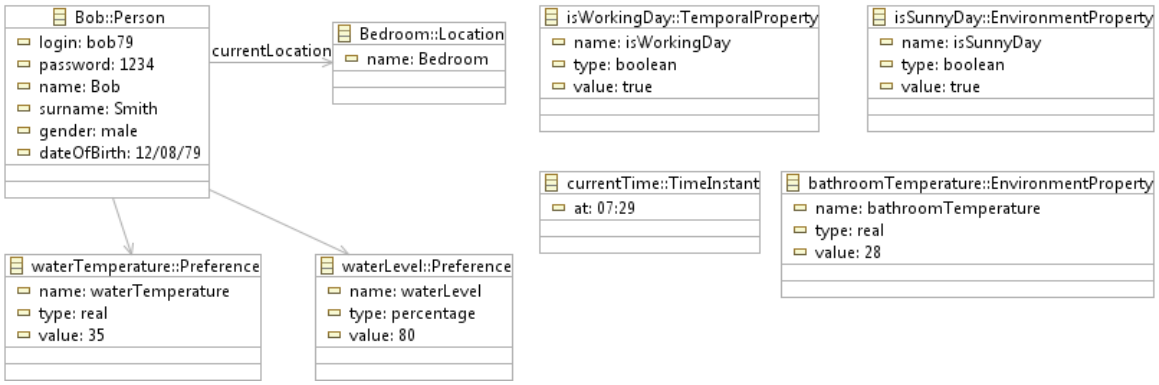

Figure 20. Ontology state in a sunny day

One minute after running the system, the context monitor updates the *currentTime* context property to 7:30, which fulfils the context situation of the *WakingUp* behaviour pattern (it is 7:30 a.m. on a working day). The behaviour pattern engine then starts the execution of the service coordination designed for the pattern. This execution trace is shown in Figure 21. This trace shows that since context precondition of the first task (turn on bathroom heating) is not satisfied, its service is not executed. After 10 minutes, the radio is turned on and the blinds are raised because it is a sunny day. Then, the bath is filled to the desired level. Finally, when Bob arrives to the kitchen, coffee is automatically made.

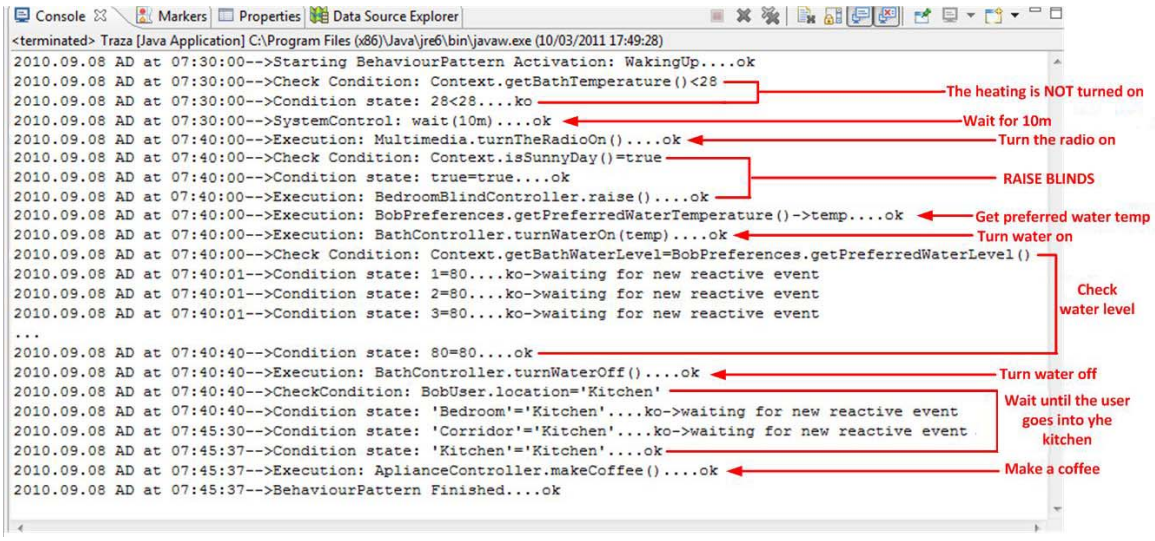

Figure 21. Service execution on a sunny day

6.2 Coordination performed on a rainy day

A partial view of the state of the context ontology is shown in Figure 22. Summarizing: Bob is in the bedroom; it is 7:29 a.m. on a rainy working day; the bathroom temperature is 22 degrees Celsius; and Bob likes having a bath with the water at 35 degrees and the bath filled to 80 percent of its capacity.

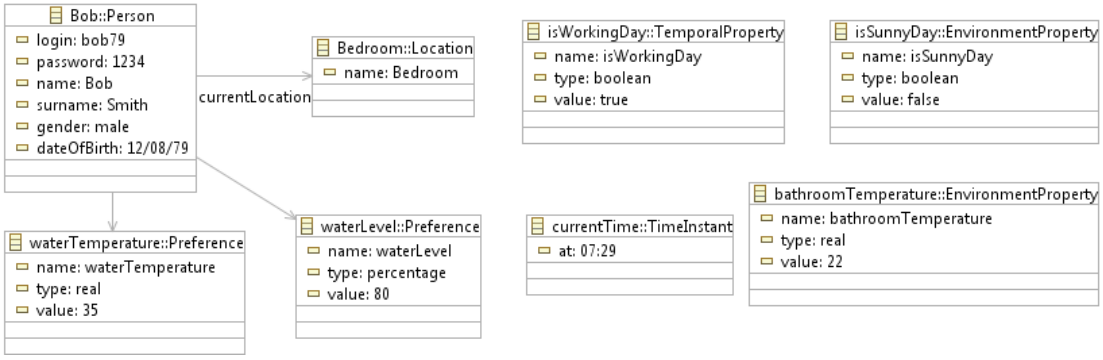

Figure 22. Ontology state in a rainy day

In the same way as in the previous state, one minute after running the system, the context monitor updates the *currentTime* context property to 7:30 and the context situation of the *WakingUp* behaviour pattern is then satisfied (it is 7:30 a.m. on a working day). The behaviour pattern engine then starts the execution of the service coordination designed for the pattern. This execution trace is shown in Figure 23. Since it is a rainy day and the bathroom temperature is colder, the trace is slightly different. Specifically, this trace shows that the bathroom heating is turned on because the temperature is less than 28 degrees (which is the context precondition of the task). In addition, after waiting 10 minutes, the radio is turned on and this time, the lights are switched on instead of raising the blinds because it is raining.

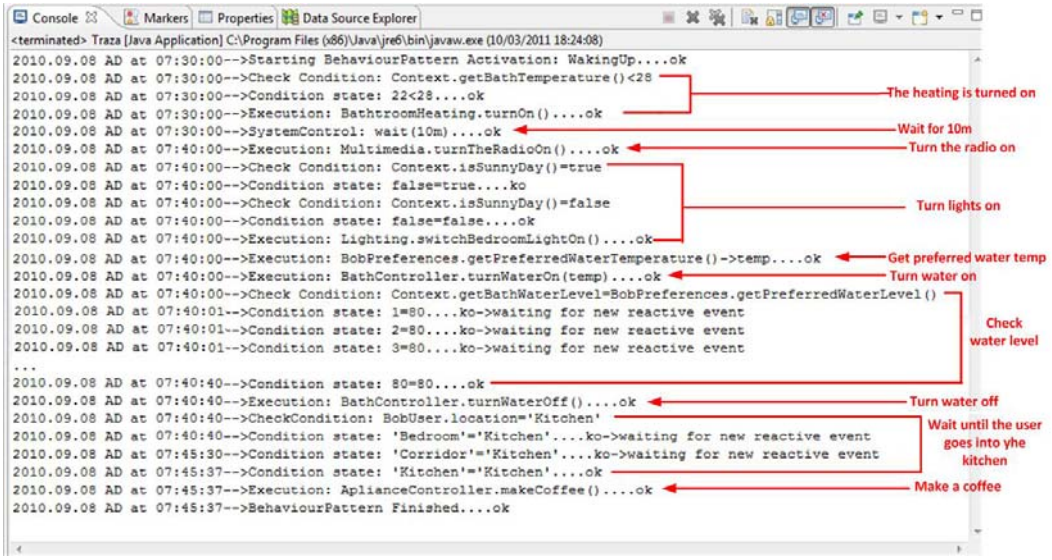

Figure 23. Service execution on a rainy day

It is important to note that, as in the previous scenario, both the execution of each task and the context changes caused by it are stored by the context monitor in the context ontology as instances of the *ExecutedOperation* and *DetectedChange* classes, respectively.

6.3 Coordination performed on Sunday

A partial view of the state of the context ontology is shown in Figure 24. Summarizing: Bob is in the bedroom; it is 7:30 a.m. on a Sunday; it is a sunny day; the bathroom temperature is 28 Celsius degrees; and Bob likes having a bath with the water at 35 degrees and the bath filled to 80 percent of its capacity.

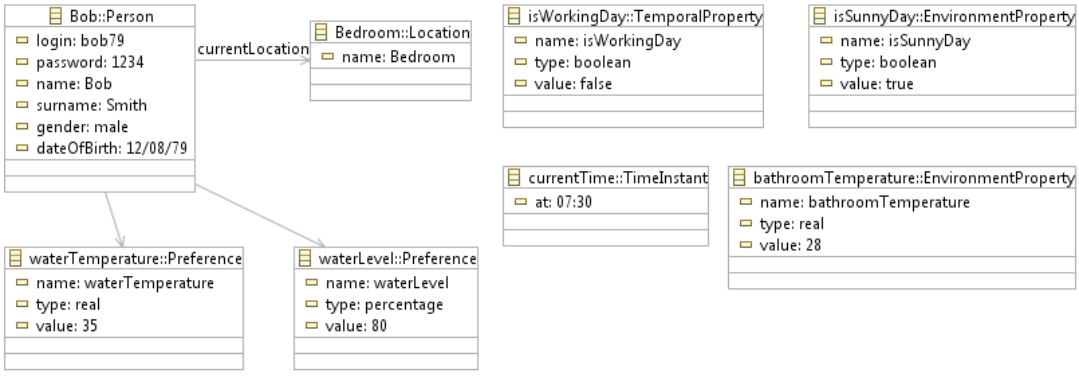

Figure 24. Ontology state on Sunday

In this context ontology state, the *WakingUp* behaviour pattern is not executed since the context situation is not fulfilled: it is not a working day because it is Sunday.

7. Discussion

In this work, we have presented an approach that allows us to define the context-adaptive coordination of pervasive services at a high level of abstraction. These abstract designs are directly executed at runtime in order to coordinate the corresponding services as designed. In this section, we discuss the main advantages of our approach and the challenges that must be confronted.

7.1. Advantages of the Presented Approach

The approach has the following main advantages:

- **Flexibility of implementation**. Our proposal allows the system modelling to remain independent of the architecture implementation. In addition, although the architecture implementation presented is based on the OSGi technology to take advantage of its facilities (such as service searching, service execution environment, service dependencies, dynamic service composition at runtime, etc.), the architecture can

be mapped to other technologies implementing the service execution environment and service searching facilities if not provided.

Furthermore, the architecture is not coupled with any other implementation aspects; therefore, it provides great flexibility to be used with any pervasive system. Pervasive services can be implemented to interact with any type of physical objects using any type of communication protocol such as KNX, Lonworks or UPnP. The only current restriction is that the pervasive services can be registered in an OSGi server.

- **Flexibility of pervasive service evolution**. The architecture runs in an OSGi server and is decoupled from service implementation (the link between tasks and services is done by service identifiers). This considerably facilitates making dynamic changes such as adding, removing, or modifying services at runtime. For instance, tools like the one presented in [45] could be developed for allowing end-users to evolve the system by adding new pervasive services at runtime. Thus, if new behaviour patterns needed new services, they could be added to the system at runtime and used for executing behaviour pattern tasks. So that a behaviour pattern task is performed by a new service, the relation between the corresponding task and that service should be established. This relation can be established by using tools such as the ones next explained.
- **Facility to provide post-deployment user adaptability tools**. We have proposed an approach to coordinate the execution of pervasive services according to the user behaviour patterns that must be supported. However, since users' daily tasks may change over time, once the system is deployed, the users may need to change the designed coordination. Note that the configured service coordination is evolved as soon as the context-adaptive task model is updated. Therefore, we can take advantage of this to develop tools that allow the end-users themselves to adapt the designed coordination after deployment.

  As a proof of concept, we have developed a toolkit that allows end-users to evolve the service coordination by means of user-friendly graphical interfaces [46]. With this toolkit, end-users can change the context conditions or adapt the designed behaviour patterns to new needs. Once the end-users have redesigned the service coordination, the toolkit modifies the task model accordingly, and the runtime service coordination is readapted. Figure 25 shows a snapshot of the toolkit, in which users can specify the context situation whose fulfilment will trigger the execution of a behaviour pattern. This toolkit has

been developed basing on Visual Programming approaches and good-practices in End-user Development [47-50].

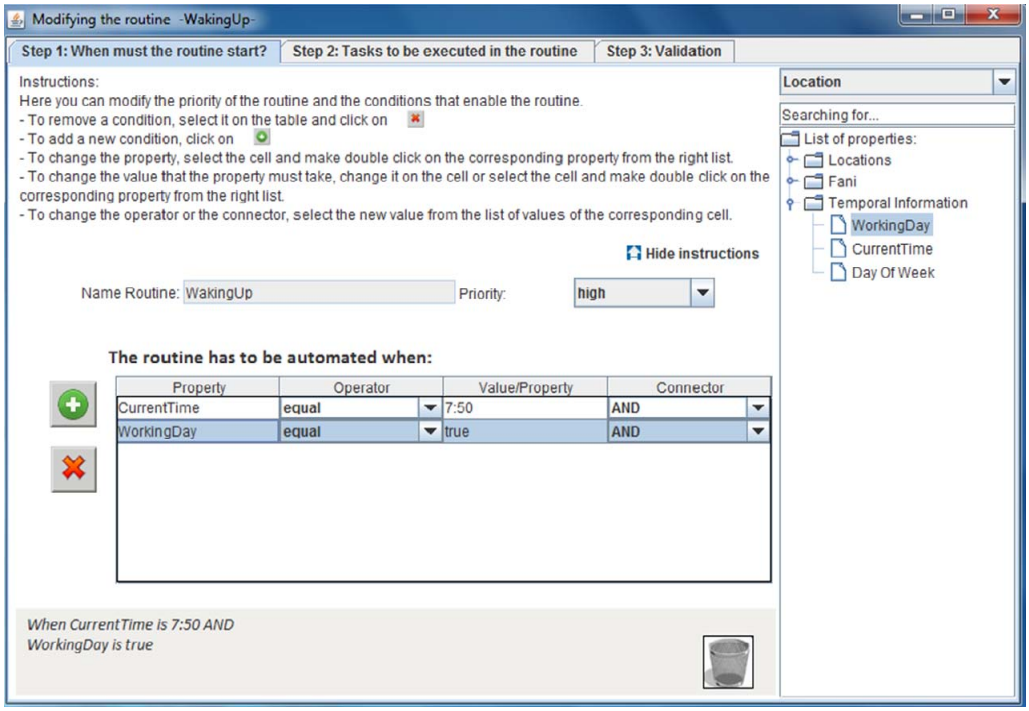

Figure 25 End-user interface for defining the context situation of the behaviour pattern

- **High-level evolution of the designed coordination**. Due to the fact that context-adaptive task models are interpreted at runtime to coordinate the execution of services, these models are the primary means to determine the services that must be executed to support user behaviour patterns. Thus, they become a key element within the evolution of this coordination since it can be evolved at runtime by simply modifying the task model. Note that as soon as the model is updated, it is reinterpreted by the Behaviour Pattern Engine and the configured service coordination is evolved.

  Thus, we can add a new coordination by creating the corresponding task hierarchy in the task model. We can also remove a specified coordination or modify it by creating new tasks, and modifying or deleting the existing ones (for instance, by using the tool presented in Section 4.2 or tools such as the one above introduced). This allows us to support the evolution of pervasive service coordination by confronting one of the most important challenges identified in software evolution: supporting evolution at higher levels of abstraction (e.g. design models) [51-53]. A video in which this evolution is performed can be seen at *www.pros.upv.es/art*.
- **Facility to change user preferences and personal data.** This advantage is closely related to the previous one. User preferences such as preferred music or temperature, and personal data such as name,

weight, or gender are stored in the context ontology. Note that the context ontology is totally decoupled from the system implementation. This provides great flexibility for changing user preferences without having to reimplement any part of the system code. This helps users to update this information at runtime. In addition, note that the coordination of services can be subject to specific user preferences making it easy to activate or deactivate a specific user behaviour pattern if user preferences change. As a proof of concept, the toolkit presented above allows users to change their preferences as well as their personal data at runtime.

7.2. Challenges to Be Confronted

Throughout the development and validation of the proposed architecture, we detected some challenges that need to be resolved in order to correctly support user behaviour patterns by service coordination. The challenges focus on the following questions:

- What happens when a behaviour pattern is not completed? In other words, what happens when a user behaviour pattern is triggered and a user action on which an intermediate task is dependent on a context constraint, is never performed or is performed when not needed?

  These situations must be avoided as much as possible. To resolve this problem we are currently extending the task model in order to allow the possibility of marking tasks as cancellable tasks, which are tasks that can be cancelled if a certain time has elapsed and the context conditions to perform them have not yet been fulfilled. For instance, in the WakingUp behaviour pattern, it may be that one day Bob is in a rush and does not go to the kitchen to have breakfast. In this case, if after one hour the user has not gone to the kitchen, the task that makes a coffee can be cancelled because it is no longer necessary.

- What happens if several behaviour patterns are activated at the same time?

  To resolve this problem we are adding priorities to user behaviour patterns. Thus, if the context situation of several behaviour patterns is fulfilled at the same time, they will be executed following their order of priority. If several behaviour patterns are activated and have the same priorities, they would be concurrently executed. However, it may be that after executing the patterns with higher priority, the context situation of some other activated pattern is not fulfilled. If so, the context situation of the activated patterns must be checked again before executing them.

- What happens if conflictive behaviour patterns are executed?

  To automate user behaviour patterns, our architecture performs the service coordination as described in the task model. If this coordination is not correctly designed, it could arise conflictive situations such as the creation of loops (i.e., an intermediate task of a behaviour pattern A triggers a behaviour pattern B, and an intermediate action of B triggers the behaviour pattern A). To facilitate and prevent these situations, we plan to extend the developed tools so that they can analyse the context-adaptive task models in order to detect these conflictive situations before they can happen. To achieve this, the services provided by the pervasive system should provide information about which operations perform contradictory tasks (e.g., switch on/off, raise/lower blinds) and also about the context properties that each service modifies. This information is required in order to know if the execution of the services coordinated in a behaviour pattern can cause another execution of the same pattern or the execution of other patterns.

- What happens if several services can be used for performing a task?

  In our approach, each leaf task specified in the task model has to be related to one service of the pervasive system that can perform it. However, if several services are available to perform the task, we could use the most appropriate service in each moment depending on the user context. For instance, for informing users about their new messages, it could be used the ambient sound service if they are alone and can listen to them, or it could be used their mobile phone display if they are not alone and cannot be disturbed at that moment. To deal with this challenge, we are extending our approach so that a task can be related to several services, describing the context conditions that make the system to choose the adequate service each moment. More detail about the work that has been developing in this direction can be found in [54].

Other extensions that we plan to include in the proposed architecture consist in supporting the use of machine-learning algorithms. We believe that these algorithms can greatly automate the evolution of service coordination. Machine-learning algorithms can use the context information captured at runtime to infer modifications or even new behaviour patterns.

8. Conclusions

In this work, we have presented and evaluated a novel approach for confronting the challenge of coordinating pervasive services in a context-adaptive way in order to satisfy user behaviour patterns. This approach confronts this challenge considering different stages of development.

We have presented a technique to describe the context-adaptive coordination of pervasive services at design time. This technique is based on the use of a task model and a context ontology, which allows us to deign context-adaptive coordination of pervasive services using concepts that are conceptually close to user behaviour patterns. This helps to validate that user behaviour patterns are properly supported by the designed service coordination.

We take these models a step further and reuse them at runtime. To do this, we have presented an architecture that automatically coordinates the execution of services according to the current context by interpreting the performed designs. This architecture provides a Behaviour Pattern Interpreter that coordinates the service execution according to task-based descriptions, and a Context Monitor that maintains a faithful representation of context in the context ontology.

Finally, the use of design models during runtime to coordinate services has several advantages for evolution. These include the possibility of performing post-deployment evolutions at runtime or performing these evolutions at a high level of abstraction.

References

1. Weiser, M. (1991) The Computer of the 21st Century. *Scientific American* 265, pp 66–75.
2. Neal, D. T. and Wood W. (2007) Automaticity in Situ: The Nature of Habit in Daily Life, ed J. A. Bargh PG, & E. Morsella (Eds.), Automaticity in Situ: The Nature of Habit in Daily Life. *Psychology of action: Mechanisms of human action,*Oxford: Oxford University Press, USA.
3. Vanderdonckt, J., Grolaux D., Roy P. V., Limbourg Q., Macq B., and Michel B. (2005) A Design Space for Context-Sensitive User Interfaces. *Proceedings of the ISCA 14th International Conference on Intelligent and Adaptive Systems and Software Engineering*, Toronto, Canada, July 20-22, pp 207-214, International Society for Computers and their Applications, Toronto.
4. France, R. and Rumpe B. (2007) Model-driven Development of Complex Software: A Research Roadmap. *FOSE '07: 2007 Future of Software Engineering*, pp 37-54, IEEE Computer Society, Seville, Spain.
5. Blair, G., Bencomo N., and France R. B. (2009) Models@run.time. *IEEE Computer,* 42, pp 22-27.
6. Henricksen, K., Indulska J., and Rakotonirainy A. (2006) Using context and preferences to implement self-adapting pervasive computing applications. *Sofware: Practice and Experience,* 36, pp 1307-1330.
7. García-Herranz, M., Haya P. A., Esquivel A., Montoro G., and Alamán X. (2008) Easing the Smart Home: Semi-automatic Adaptation in Perceptive Environments. *Journal of Universal Computer Science,* 14, pp 1529-1544.
8. Morin, B., Fleurey F., Bencomo N., Jézéquel J.-M., Solberg A., Dehlen V., and Blair G. (2008) An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability. *11th ACM/IEEE*

International Conference on Model Driven Engineering Languages and Systems (MoDELS'08), Toulouse, France, 28 September - 3 October.

9. Bencomo, N., Grace P., Flores-Cortes C., Hughes D., and Blair G. (2008) Genie: supporting the model driven development of reflective, component-based adaptive systems. *ICSE'08*, Leipzig, Germany, 10 - 18 May 2008, pp 811-814, ACM New York, New York, USA.
10. Blumendorf, M., Lehmann G., Feuerstack S., and Albayrak S. (2008) Executable Models for Human-Computer Interaction. *International Workshop on the Design, Verification and Specification of Interactive Systems (DSV-IS)*, Kingston, Ontario,Canada, July 16-18, pp 238-251, Springer-Verlag Berlin.
11. Li, J., Bu Y., Chen S., Tao X., and Lu J. (2006) FollowMe: On Research of Pluggable Infrastructure for Context-Awareness. *20th International Conference on Advanced Information Networking and Applications (AINA 2006)*, Vienna, Austria, 18-20 April, pp 199-204, IEEE Computer Society.
12. Cho, Y., Shin K., Choi J., and Choi J. (2007) A Context-Adaptive Workflow Language for Ubiquitous Computing Environments. *Computational Science and Its Applications- ICCSA 2007*, pp 829-838, Springer-Verlag Berlin Heidelberg.
13. Dey, A. K. (2001) Understanding and Using Context. *Personal Ubiquitous Computing*.
14. H. Chen, Finin T., and Joshi A. (2004) An ontology for context-aware pervasive computing environments. *Special Issue on Ontologies for Distributed Systems, Knowledge Engineering Review*, 18, pp 197–207.
15. W. N. Schilit, Adams N. I., and Want R. (1994) Context-aware Computing Applications. *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, February 23-24, pp 85-90, IEEE Computer Society, Santa Cruz, California, USA.
16. Ou, S., Georgalas N., Azmoodeh M., Yang K., and Sun X. (2006) A Model Driven Integration Architecture for Ontology-based Context Modelling and Context-Aware Application Development. *ECMDA-FA*, Bilbao, Spain, July 10-13, Lecture Notes in Computer Science, Springer, Berlin.
17. Bardram, J. E. (2005) The Java context awareness framework (JCAF) - a service infrastructure and programming framework for context-aware applications. *Third International Conference on Pervasive Computing*, Munich, Germany, May 8-13, pp 98–115, Lecture Notes in Computer Science, Srpinger, Berlin.
18. Henricksen, K. and Indulska J. (2006) Developing context-aware pervasive computing applications: Models and approach. *Pervasive and Mobile Computing*, 2, pp 37-64.
19. Sheng, Q. Z. and Benatallah B. (2005) ContextUML: a UML-based modelling language for model-driven development of context-aware web services. *ICMB'05*, 11-13 July, pp 206 - 212 IEEE Computer Society, Washington, DC, USA.
20. Coutaz, J., Crowley J. L., Dobson S., and Garlan D. (2005) Context is Key. *COMMUNICATIONS OF THE ACM*, 48, pp 49-53.
21. Mokhtar, S. B., Kaul A., Georgantas N., and Issarny V. (2006) Efficient Semantic Service Discovery in Pervasive Computing Environments. *ACM/IFIP/ USENIX 2006*, Melbourne, Australia, Nov. 27–Dec, pp 240-259, Springer-Verlag New York, Inc., New York, NY, USA.
22. Henricksen, K. and Indulska J. (2004) A Software Engineering Framework for Context-Aware Pervasive Computing. *PerCom*, Orlando, Florida 14-17 March, p 77, IEEE Computer Society, Washington, DC, USA.
23. Lewis, G. A. and Smith D. B. (2008) Service-Oriented Architecture and its implications for software maintenance and evolution. *FoSM* Beijing, China, September 28 to October 4, pp 1 - 10
24. Baldauf, M., Dustdar S., and Rosenberg F. (2007) A Survey on Context-Aware Systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2, pp 263-277.
25. Ye, J., Coyle L., Dobson S., and Nixon P. (2007) Ontology-based models in pervasive computing systems. *The Knowledge Engineering Review*, 22, pp 315-347.
26. Thompson, M. and Midkiff S. (2005) Service description for pervasive service discovery. *ICDCSW'05*, Columbus, Ohio, USA, 6th June, pp 273-279, IEEE Computer Society, Washington, DC, USA.
27. Stevenson, G., Knox S., Dobson S., and Nixon P. (2009) Ontonym: A Collection of Upper Ontologies for Developing Pervasive Systems. *CIAO '09 Proceedings of the 1st Workshop on Context, Information and Ontologies, at the 6th European Semantic Web conference (ESWC 2009)*, Heraklion, 31 May - 1st June, pp 1-8, ACM New York, NY, USA.
28. Gu, T., Pung H. K., and Zhang D. Q. (2005) A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, 28, pp 1-18.
29. Johnson, P. (1999) Tasks and situations: considerations for models and design principles in human computer interaction. *HCI International*, Munich, Germany, 22-27 August, pp 1199–1204, Lawrence Erlbaum Associates.

30. Limbourg, Q. and Vanderdonckt J. (2004) Comparing Task Models for User Interface Design, eds Diaper D & Stanton NA (Eds.), Comparing Task Models for User Interface Design. *The Handbook of Task Analysis for Human-Computer Interaction*,Lawrence Erlbaum Associates, Mahwah, NJ.
31. Lauesen, S. (March-April 2003) Task Description as Functional Requirements. *IEEE Software* 20, pp 58-65.
32. Paternò, F. (2002) ConcurTaskTrees: An Engineered Approach to Model-based Design of Interactive Systems. *The Handbook of Analysis for Human-Computer Interaction*, pp 483-500 Lawrence Erlbaum Associates.
33. Pribeanu, C., Limbourg Q., and Vanderdonckt J. (2001) Task Modelling for Context-Sensitive User Interfaces. *International Workshop on the Design, Verification and Specification of Interactive Systems (DSV-IS)*, Glasgow, Scotland, UK, pp 49-68, Springer-Verlag Berlin Heidelberg.
34. Vanderdonckt, J. (2005) A MDA-Compliant Environment for Developing User Interfaces of Information Systems. *CAiSE 2005*, Porto, Portugal, June 13-17, pp 16-31, Lecture Notes in Computer Science 3520 Springer
35. Huang, R., Cao Q., Zhou J., Sun D., and Su Q. (2008) Context-Aware Active Task Discovery for Pervasive Computing. *International Conference on Computer Science and Software Engineering*, Wuhan, China, December 12-14, IEEE Computer Society.
36. Sousa, J., Poladian V., Garlan D., and Schmerl B. (2006) Task-based Adaptation for Ubiquitous Computing. *IEEE Transactions on Systems, Man, and Cybernetics*, 36, 3, pp 328-340.
37. Hoyos, J. R., García-Molina J., and Botía J. A. (2010) MLContext: A Context-Modeling Language for Context-Aware Systems. *Proceedings of the Third International DisCoTec Workshop on Context-Aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS 2010)*, Amsterdam, Netherlands, 10th June, pp 1-14, Electronic Communications of the EASST.
38. Shepherd, A. (2001) *Hierarchical Task Analysis*, Taylor & Francis, London.
39. Turner, K. J. (1996) The Formal Specification Language LOTOS: A Course for Users. University of Stirling.
40. Loke, S. W. (2009) Building Taskable Spaces over Ubiquitous Services. *IEEE Pervasive Computing*, 8, pp 72-78.
41. Brossard, A., Abed M., and Kolski C. (2011) Taking context into account in conceptual models using a Model Driven Engineering approach. *Information and Software Technology*, 53, pp 1349-1369.
42. Völter, M. (2006) Software Architecture - A pattern language for building sustainable software architectures. *In EuroPLoP' 2006, Eleventh European Conference on Pattern Languages of Programs*, Irsee, Germany, July 5-9, pp 31-66, UVK - Universitaetsverlag Konstanz.
43. Ruiz, M. (2010) Generación automática de servicios web a partir de modelos conceptuales. (Universidad Politécnica de Valencia, Valencia).
44. Sirin, E., Parsia B., Grau B. C., Kalyanpur A., and Katz Y. (March 2007) Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5, pp 51-53.
45. Weis, T., Handte M., Knoll M., and Becker C. (2006) Customizable Pervasive Applications. *4th IEEE International Conference on Pervasive Computing and Communication (PerCom)*, Pisa, Italy, 13-17 March, pp 239-244, IEEE Computer Society.
46. Serral, E., Pérez F., Valderas P., and Pelechano V. (2010) An End-User Tool for Adapting Smart Environment Automation to User Behaviour at Runtime. *4th Symposium of Ubiquitous Computing and Ambient Intelligence, UCAmI 2010*.
47. Welie, M. v. and Trætteberg H. (2000) Interaction Patterns in User Interfaces. *Seventh Pattern Languages of Programs Conference (PLoP 2000)*, Illinois, USA, 13-16 August, pp 13-16.
48. Pérez, F. and Valderas P. (2009) Allowing End-users to Actively Participate within the Elicitation of Pervasive System Requirements through Immediate Visualization. *Fourth International Workshop on Requirements Engineering Visualization (REV)*, Atlanta, Georgia, USA 1 September, pp 31 - 40 IEEE Computer Society, Washington, DC, USA.
49. Lieberman, H., Paternó F., and Wulf V. (2006) *End User Development*, Springer,
50. Galitz and O. W. (2002) *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*, John Wiley \& Sons, Inc., New York, NY, USA.
51. Mens, T. (2009) The ERCIM Working Group on Software Evolution: the Past and the Future. *IWPSE-Evol'09*, Amsterdam, the Netherlands, pp 1-4, ACM, New York, NY, USA.
52. Ajila, S. A. and Alam S. (2009) Using a Formal Language Constructs for Software Model Evolution. *Third IEEE International Conference on Semantic Computing (IEEE-ICSC 2009)*, Berkeley, CA, USA, September 14-16, pp 390-395, IEEE Computer Society, Washington, DC, USA.
53. Bennett, K. and Rajlich V. (2000) Software Maintenance and Evolution: A Roadmap. *22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 4-11, pp 75–87, ACM.

54. Gil, M., Serral E., Valderas P., and Pelechano V. (2011) Achieving Unobtrusive Interaction for Routine Task Automation. *Proceedings of the V International Conference on Ubiquitous Computing and Ambient Intelligence (UCAmI).* December 2011, Ribiera Maya.