# Supporting Runtime System Evolution to Adapt to User Behaviour*

Estefanía Serral, Pedro Valderas, and Vicente Pelechano

Centro de Investigación en Métodos de Producción de Software (ProS)
Universidad Politécnica de Valencia, Valencia, Spain
{eserral,pvalderas,pele}@dsic.upv.es

**Abstract.** Using a context-aware approach, we deal with the automation of user routines. To do this, these routines, or user behaviour patterns, are described using a context model and a context-adaptive task model, and are automated by an engine that executes the patterns as specified. However, user behavior patterns defined at design time may become obsolete and useless since users needs may change. To avoid this, it is essential that the system supports the evolution of these patterns. In this work, we focus on supporting this evolution by confronting an important challenge in evolution research: raise the level in which evolution is applied to the modelling level. We develop mechanisms to support the pattern evolution by updating the models at runtime. Also, we provide end-users with a tool that allows them to carry out the pattern evolution by using user-friendly interfaces.

**Keywords:** Context adaptation, models, tasks, user behaviour pattern automation.

## 1 Introduction

Context-aware systems are those capable of adapting its behaviour according to context and performing actions on behalf of users without being intrusive. Using a context-aware approach, we deal with the automation of user routines. A routine, or behaviour pattern, is a set of tasks characterized by habitual repetition in similar contexts [1]. Some patterns are determined by our lifestyle, e.g. reading electronic mail and opening certain web pages as soon as we have access to internet; others are reactions to things happening around us, e.g. lowering every blind and winding up every awning when it starts to rain.

Several works [2, 3] have dealt with inferring these patterns from user action observation by using machine-learning algorithms. However, they require a lot of training data and may automate actions that users do not want to be automated because algorithms do not know the semantics of user-performed actions. Moreover, note that many patterns are well known before the system is implemented. These patterns can

* This work has been developed with the support of MEC under the project SESAMO TIN2007-62894 and co financed by FEDER, in the grants' program FPU.

B. Pernici (Ed.): CAiSE 2010, LNCS 6051, pp. 378–392, 2010.
© Springer-Verlag Berlin Heidelberg 2010

be described at design time and automated under the adequate context conditions. Our approach [22] achieves this. It presents a context-adaptive task model that allows each behaviour pattern to be specified through a set of tasks that have to be carried out to automate it. These tasks are modelled according to context, which is previously specified in a context model. Both the context model and the task model are also used at runtime. A Model-based Automation Engine (MAtE) interprets them to execute the patterns as specified.

However, this is not enough to automate user behaviour patterns since user needs may change in the future. It is essential that the system supports the evolution of the designed patterns to adapt to user behaviour changes; otherwise, their automation may become a burden on users instead of a way of helping them. In this work, we focus on supporting this evolution. To do this, we confront one of the most important challenges identified in software evolution according to works such as [4, 5]: raise the level of abstraction in which evolution is applied to the modelling level, i.e. perform system evolution by evolving the models that specify it. Since the research in model-driven development started, this challenge becomes increasingly more relevant, and new approaches and tools for dealing with it are urgently needed. However, although many advances have been done in software evolution research, almost all existing approaches for supporting it are primarily targeted to source code, including the proposed approaches for automating user routines that supports evolution. In contrast, we propose applying runtime evolution at modelling level, which allows us to manage it at a high level of abstraction.

To do this, we take advantage of the design of our approach. Since the task model and the context model are interpreted at runtime to execute the corresponding behaviour patterns, as soon as the models are modified, the changes are applied by the system. Thus, we design and implement mechanisms to support the update of both models at runtime [6] using the same high level concepts used for building the models. Furthermore, we provide end-users with a user-friendly tool to evolve the patterns. This end-user tool uses the provided mechanisms to carry out the evolution. Note that these mechanisms could also be automatically applied, for instance, by using machine-learning algorithms that detect changes in user behaviour. However, this may lead to applying changes that do not correspond to user desires and to automating tasks that users do not want to be automated. This may be annoying to users and may cause users to feel loss of control of the system.

The remainder of the paper is organized as follows. Section 2 gives an overview of our approach. Section 3 describes real examples of behaviour patterns. Section 4 describes the context model and the task model. Section 5 presents the mechanisms to address pattern evolution. Section 6 explains the end-user tool. Section 7 evaluates our approach. Section 8 presents the related work. Section 9 presents the conclusions and the future work.

## 2 Automating User Behaviour Patterns: An Overview

Our approach, which is shown in Fig. 1, deals with automating the routines that users want to be automated, according to their desires and demands. To achieve this, analysts first interview end-users to determine the tasks that they perform and identify

behaviour patterns in these tasks. Also, the analysts determine the context in which the pattern has to be triggered and the context in which its tasks are performed. Thus, the analysts describe the needed context information by using a context model. With the participation of users, the analysts then specify the context situation that triggers the execution of the pattern, the tasks to be executed for each one of the identified patterns, and the temporal relationships that must be accomplished for the execution of these tasks. The behaviour pattern tasks and their relationships are specified using the context information of the context model, in such a way that the task execution automatically adapt according to context.

In addition, the tasks to be executed are related to a pervasive service that can carry it out. A pervasive service is a piece of software that is in charge of controlling devices in order to achieve a task goal. For instance, if a task whose goal is to lower the blinds has been defined, this task is associated to a pervasive service that executes this action by interacting with the blind device. These services are also in charge of updating the context model when context changes arise (e.g. indicating the current state of blinds: open or closed). To implement these pervasive services, a MDD method that generates them in Java/OSGi [7] technology from models is used. As an example, Figure 1 shows a partial view of a Java/OSGi-based pervasive service implementation. This method is out of the scope of this paper; more information about it can be found in previous works [8, 9].

Once the behaviour patterns have been specified using the Context Model and the Task Model, a Model-based user task Automation Engine (MAtE) is in charge of automating the patterns when needed. To do this, MAtE uses a Context Monitor that monitors the context changes reflected by changes on the context model. When a context change is arisen, MAtE checks whether some context situations are fulfilled by interpreting the task model and querying the context model at runtime. If so, MAtE interprets the task model to perform the tasks of the corresponding pattern according to their specification. To perform each task, MAtE executes the pervasive service related to it.

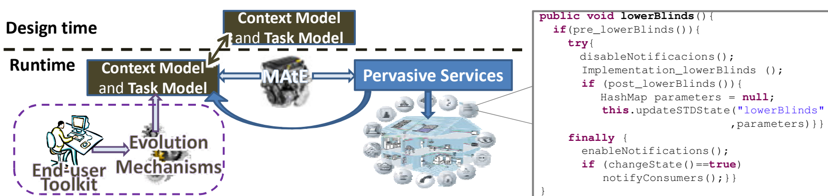



**Fig. 1.** Overview of the approach

However, without explicit support for pattern evolution, system may become progressively less satisfactory. In this work, we deal with this problem by allowing end-users to be able to evolve the behaviour patterns according to their needs. To do this, our approach provides a set of mechanisms to allow behaviour patterns to be evolved by updating the task model and the context model at runtime. Moreover, we provide end-users with a tool that allows them to evolve these patterns by using user-friendly interfaces. This tool uses the provided mechanisms to carry out the evolution.

## 3 User Routines: A Case Study

We present some user examples in the smart home domain, since it is close to readers. A married couple (we will refer to them as Bob and Sarah), was first interviewed to determine their domestic behaviour patterns. We next present four representative examples of the behaviour patterns identified for the case study:

1. **WakingUp:** At 7.00 a.m. on working days, the system wakes Bob up with his preferred music and raises the bedroom blinds to the middle.
2. **GoingOut:** When Bob and Sarah left the house, the system closes all the windows and doors, and switches off all the lights. If it is a working day, the system puts the air and heating conditioner in energy-saving mode; otherwise, the system switches it off, turns the water off at the mains and turns the gas off. Finally, the system reminds users to enable security (they prefer to activate the security themselves; therefore, the system only sends them a notification to remind them to do this).
3. **Recording:** If Sarah is not at home at 18.30 p.m., her favourite program is recorded.
4. **StormSecurity:** If it starts to rain, the system lowers all the blinds and winds up all the awnings. If it does not stop raining and the garden sprinklers are switched on, the system switches them off. When it stops raining, the system calculates the cubic meters of rainfall and updates the irrigation timetable according to it.

## 4 The Context Model and the Task Model

The context model and the task model are used at design time for specifying the behaviour patterns that users want to be automated and their context; and also at runtime for supporting the specified behaviour pattern automation and evolution.

### 4.1 Ontology-Based Context Model

The context model semantically describes the Context required for properly automating the user behaviour patterns. This model is based on an ontology proposed in previous works [8], which defines classes such as *User*, *EnviromentProperty* or *Location* to capture context. We used an ontology-based approach because it provides a formal analysis of the domain knowledge and allows common understanding of the structure of context. At design time, we use the EODM plugin [10] to graphically specify the context. EODM provides a tree graphic editor and also stores the model in the Web Ontology Language (OWL) [11]. OWL is an ontology language that greatly facilitates knowledge automated reasoning and is a W3C standard. At runtime, we use the model representation in OWL. In OWL, the classes of the ontology are defined by OWL classes, and the context of the system is represented by OWL individuals, which are instances of these classes. Fig. 2 shows an example of context model in its both representations (design and run time).

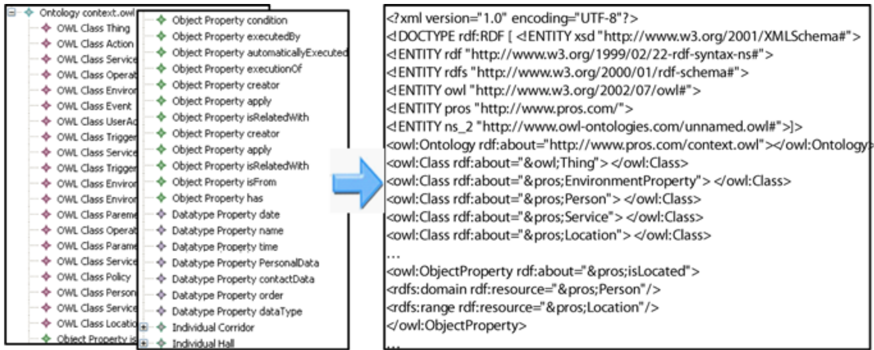



**Fig. 2.** An example of context model (graphical and OWL representation)

## 4.2 Context Adaptive Task Model

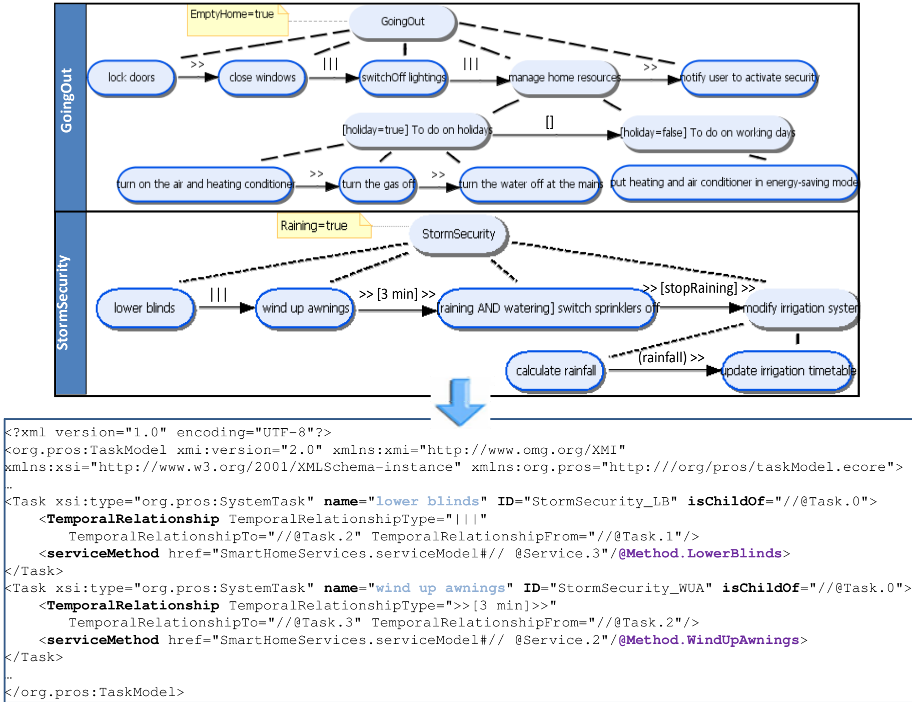
This model describes the behaviour patterns that have to be automated. Each pattern is specified by a **task hierarchy**. The root task represents the behaviour pattern and has an associated context situation, which defines the set of context conditions whose fulfilment triggers the execution of the pattern. It is broken down into *Composite Tasks* (which are intermediate tasks that can be broken down), and/or *System Tasks* (which are leaf tasks). Composite Tasks are used for grouping subtasks that share a common behaviour or goal. System tasks can be supported by a pervasive service. Both types of task can have a context precondition, which defines the context conditions that must be accomplished so that a task is performed (if the precondition is not accomplished, the task is not executed). Each task also inherits the context preconditions of its parent task.

As examples, the modelling of the *GoingOut* and *StormSecurity* patterns (Section 3) using the task model is shown at the top of Fig. 3. For instance, the *StormSecurity* behaviour pattern is triggered when it starts to rain as indicated in the context situation. The pattern is split into four tasks: the *lower blinds, wind up awnings* and *switch sprinklers off* System Tasks, and the *modify irrigation system* Composite Task, which is also split into the *calculate rainfall* and *update irrigation timetable* System Tasks. In addition, the *switch sprinklers off* System Task has a context precondition (which is shown between brackets before the name of the task) that indicates that this task will be only executed when it is raining and the system is watering the garden.

Tasks with the same father task are related by **temporal relationships**. They describe how the tasks are executed. For instance, the temporal relationship between the *wind up awnings* and *switch sprinklers off* tasks indicates that the sprinklers will be switched off 3 minutes after the awnings have been wound up (provided the context precondition fulfils).

At design time, the task model is specified by means of an editor developed using the Eclipse platform and the EMF and GMF plugins [10]. By using this editor, the model can be graphically edited (as the top section of Fig. 3 shows) and is also stored in XMI (XML Metadata Interchange). At runtime, we use its representation in XMI. The bottom section of Fig. 3 shows part of the XMI representation of the

*StormSecurity* pattern, where the properties of the *lower blinds* and *wind up awnings* tasks are shown. Note that the *service* property is a reference to the service in charge of executing the task; e.g. the *lower blinds* task is related to the *lowerBlinds* method, which was presented in Section 2.




```
<?xml version="1.0" encoding="UTF-8"?>
<org.pros:TaskModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:org.pros="http:///org/pros/taskModel.ecore">
..
<Task xsi:type="org.pros:SystemTask" name="lower blinds" ID="StormSecurity_LB" isChildOf="//@Task.0">
    <TemporalRelationship TemporalRelationshipType="|||"
       TemporalRelationshipTo="//@Task.2" TemporalRelationshipFrom="//@Task.1"/>
    <serviceMethod href="SmartHomeServices.serviceModel#// @Service.3"/@Method.LowerBlinds>
</Task>
<Task xsi:type="org.pros:SystemTask" name="wind up awnings" ID="StormSecurity_WUA" isChildOf="//@Task.0">
    <TemporalRelationship TemporalRelationshipType=">>[3 min]>>"
       TemporalRelationshipTo="//@Task.3" TemporalRelationshipFrom="//@Task.2"/>
    <serviceMethod href="SmartHomeServices.serviceModel#// @Service.2"/@Method.WindUpAwnings>
</Task>
..
</org.pros:TaskModel>
```

**Fig. 3.** Examples of behaviour pattern modelling (graphical and XMI representation)

## 5 Addressing the Evolution of User Behaviour Automation

The design of our approach facilitates to perform system evolution at modelling level, which is one of the top challenges in software evolution research. This is because MAtE automates the patterns by interpreting the models (the context model and the task model) where they have been specified, at runtime. Thus, if the models are changed to adapt the patterns, the changes are also taken into account by MAtE. Therefore, our approach gives us three immediate benefits to perform system evolution: 1) we do not have to maintain the consistency between the system modelling and system implementation as modifications are applied; 2) the evolution can be managed at a high level of abstraction (modelling level); 3) models can provide us with a richer semantic base for runtime decision-making related to system adaptation.

From these premises, we address the pattern evolution by adapting the models where they are specified. We provide mechanisms capable of evolving the task model and the context model at runtime. These mechanisms use the concepts of the own

language (task, behaviour pattern, user, etc.) used for specifying the models, which facilitates understand and handle the evolution. In addition, the mechanisms ensure that the changes are in accordance with their metamodel definition and, therefore, syntactically correct. In this way, these mechanisms define how the patterns can be changed over time and maintain software quality characteristics. Note that the use of such mechanisms raises the evolution level to the modelling level which allows the system to be evolved at runtime by using high level abstraction concepts instead of by changing lines of code. Furthermore, the mechanisms are implemented in Java applying software design patterns [21] and are provided as APIs; therefore, they can be imported and used by any Java application.

### 5.1 Supporting the Task Model Evolution

Each automated behaviour pattern is specified by a task hierarchy in the context-adaptive task model. Thus, to support the system adaptation to new user automation requirements, we have implemented a set of Model-Based User Task evolution mechanisms (MUTate) that allows evolving these patterns. For instance, MUTate allows the following: adding new tasks to a pattern; modifying the context precondition that must be accomplished so that a task can be executed; creating a new behaviour pattern; etc. To do this, MUTate provides an API that allows any elements of the specified task model (which are those specified in its metamodel, such as Behaviour Patterns, Tasks, Relationships between tasks, etc.), to be created, modified, or deleted. Specifically, this API consists of a Java class for each one of the elements of the task model metamodel. Each class provides:

- An attribute for each one of the properties and relationships of the metamodel element that the class represents; e.g., the *BehaviourPattern* class has *name* and *task* as attributes.
- Get, set and delete methods for each one of these attributes; e.g., *getName*.
- An add method for the attributes whose type is a List. This method allows an element to be directly added to the list; e.g., *addTask* method.
- Get and delete methods for access to a certain element of the attributes whose type is a list. These methods allow us to get and delete an element of the list by searching for it by using one of its identifier properties; e.g. get*TaskByID.*

In addition, the API provides a *Factory* class for creating new instances (of the classes defined in the task metamodel) of a task model.

We have used the EMF, EMF Model Query (EMFMQ), and EMF Model Transaction (EMFMT) plugins of the Eclipse Platform [10] to implement MUTate. From the metamodel of the task model in ecore, EMF generates a basic API for managing a task model. This API provides the *Factory* and *Model* classes, as well as a Java interface and an implementation class for each one of the classes of the metamodel. These implementation classes provide get and set methods to access and change the information of the instances specified in the model. We have extended these classes by implementing the methods explained above. EMFMQ is used to search for and get the instances of the model that need to be modified. EMFMT provides us with mechanisms for making transactions, reading and writing models on multiple threads,

and validating the semantic integrity of the modified model by detecting invalid changes. We have also extended the implementation provided classes in order to: 1) add, modify, and delete a complete behaviour pattern as a unique transaction; 2) allow the reading and writing of the task model at the same time; and 3) semantically validate the changes in the task model.

Figure 4 shows a partial view of the source code of MUTate. As an example, the figure shows the *getBehaviourPatternByID* method of the *TaskModel* class. This method returns the behaviour pattern whose ID is the same that the *behaviourPatternID* argument value. To find the pattern, it searches for it by using a query statement build with EMFMQ.

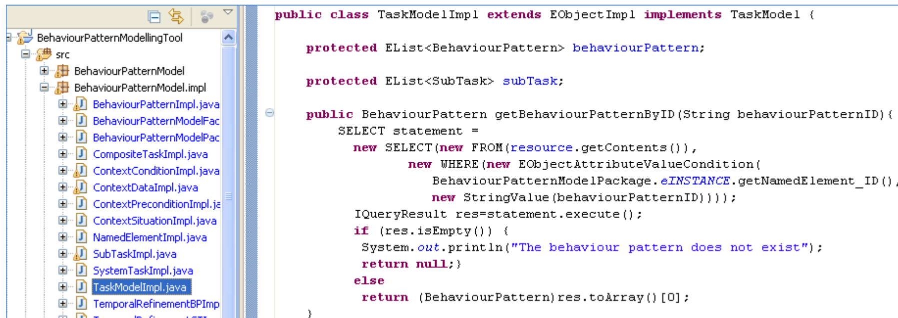


**Fig. 4.** Partial view of MUTate source code

## 5.2 Supporting the Context Model Evolution

New automation requirements may require new context information to support them. In our approach, Context is captured in the OWL context model as OWL individuals. To change the context model, we have implemented a set of Ontology-based Context model Evolution mechanisms (OCEan) that allows evolving these individuals. For instance, if a change in a pattern requires knowing the preferred environmental temperature of Bob and Sarah, OCEan allows us to create this new user preference (e.g. *idealTemperature*) to the User individuals that represent Bob and Sarah. To do this, OCean provides an API that allows any individual of the context model to be created, obtained, modified, and deleted. Specifically, this API consists of a *Model* class that allows us to open the context model, save it, and manage its individuals (*addIndividual*, *getIndividual* and *deleteIndividual*) in a generic way. The API also provides an implementation class (and its Java interface) for each one of the OWL classes defined in the context ontology. Each Java class provides:

- An attribute for each one of the properties and relationships of its OWL class; e.g., the *User OWL* class has *DNI* and *preference* as attributes.
- Get and set methods for each one of these attributes; e.g., *getDNI*.
- An add method for the attributes whose type is a List. This method allows an element to be directly added to the list; e.g., *addPreference* method.

- Get and delete methods for access to certain element of the attributes whose type is a List. These methods allow us to get or delete an element of the list by searching for it using one of its identifier properties; e.g. *getPreferenceByName*.
- The new method that creates an individual of the corresponding OWL class and calls the *addIndividual* method of the Model class to add it to the context model.

We have used the OWL API 2.1.1 [11], SPARQL [12] and the Pellet reasoner 1.5.2 [13] to implement OCean. The OWL API is an open-source API that provides facilities for creating, examining and modifying any element of an OWL ontology. SPARQL is a graph-matching query language recommended by the W3C that allows queries to be built to search for certain individual. Pellet is an open-source OWL reasoner that provides reasoning services. Pellet allows us to launch a SPARQL query against the model.

Figure 5 shows a partial view of the source code of OCean. As an example, the figure shows the *deleteIndividual* method of the *ContextModel* class. Using the OWL API, this method first creates a *remover* entity, it then gets the individual that is to be deleted, and passes it to the *remover* entity for deletion. Finally, the method applies the changes in the context model.

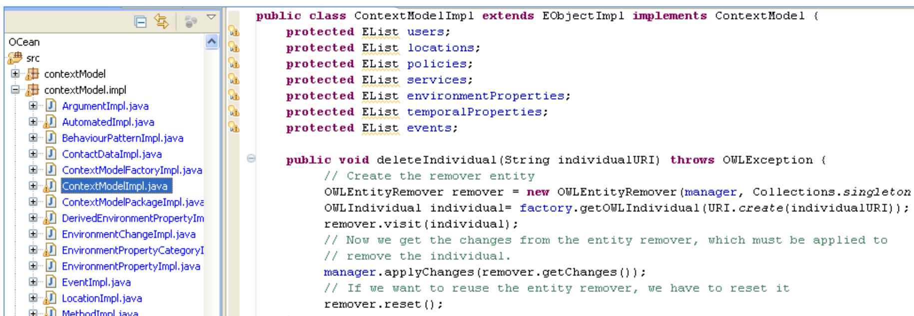



**Fig. 5.** Partial view of OCean source code

## 5.3 Using the Evolving Mechanisms

In this subsection, we present an example of how the presented mechanisms are used (see Figure 6). In this example, we add a new preference (*wakeUpTime*) to the user Bob. This preference indicates the time he prefers to wake up. To do this, we simply create a new preference, search for the user that represents Bob and add the created preference to his list of preferences. Afterwards, we change the context situation of the *WakingUp* pattern in order to indicate that it must be triggered when the time is the value indicated in the *wakeUpTime* preference. Thus, MUTate and OCEan allow the models to be evolved by using concepts of a high level of abstraction (Preference, ContextSituation, etc.), which are easy for developers to understand and use. However, these mechanisms are still difficult for end-users. For this reason, we have also developed an end-user tool that allows them to evolve the patterns by using intuitive interfaces.

```
//Create a new user preference
Preference preference= new Preference("wakeUpTime","8:00");
User user= contextModel.getUserByName("Bob");
user.addPreference(preference);
//Using this preference in the context situation of a pattern
BehaviourPattern behaviourPattern = taskModel.getBehaviourPattern("WakingUp");
behaviourPattern.setContextSituation("currentTime=wakeUpTime AND workingDay=true");
```

**Fig. 6.** Code example for model evolution

## 6 End-User Toolkit for Evolving the System

To carry out the pattern evolution, we provide end-users with a tool that allows them to use MUTate and OCEan in a user friendly way. This tool provides users with the following functionalities:

- Context Specification: the tool shows users the context information for which they have permission. It also allows a user to add new individuals corresponding to his/her information, modify them, and delete them if they are not used in the task model.
- Pattern Specification: the tool allows users to add, modify, or delete behaviour patterns by facilitating the information necessary to do this. If users do not want certain patterns to be executed during a period of time, they also can enable or disable specified patterns.

To provide user interfaces with this functionality, we have been inspired by the *Natural Programming* and *Visual Programming* end-user development approaches [14]. Based on these approaches, we have developed an interface for each one of the provided functionalities by using the SWT (Standard Widget Toolkit) [10] plugin in Eclipse. All of them follow a similar style. At the top of the interface, we guide users by using tabs that indicate the previous, current, and next steps to perform in order to achieve the corresponding goal. At the bottom of the interface, a text message is shown where help and explanations are provided to end-users. The rest of the interface is divided into two main frames: the left frame, which represents the work area where users specify the corresponding information; and the right frame, which shows users the information that they need for each step. Thus, end-users just need to select the information from the right frame and drag it to the proper location in the left frame.

Figure 7 shows a snapshot of an interface of our prototypical tool. This snapshot shows the first step for creating a new behaviour pattern in the task model: the specification of the context situation whose fulfilment triggers the execution of the pattern. The current tab and the other tabs that allow the user to navigate through the steps to be accomplished are shown at the top of the interface. On the right side, the context information that is available for Bob (who is using the interface) is shown in a tree form, going from more general information to more specific. In the work area, we facilitate the needed operators to form the context situation. Finally, the information area is provided at the bottom of the interface. This area shows what the pattern will do in a language close to natural language.

By using these interfaces, end-users can carry out the changes that they need. However, to preserve software quality characteristics, these changes are validated before they are applied to the system. Up to date, these changes are revised by

analysts; however, we are currently developing a tool to ensure the reliability of the system without the participation of analysts. This tool generates all the possible context situations that may play a part in the behaviour patterns changed by users. Then, it builds a set of tests that have to be passed by the system before these changes can be taken into account. If any of these tests are not passed, the system notifies the users about the possible mistakes so that they can be corrected. Finally, once the changes are validated, the tool updates the task model and the context model accordingly. The tool uses MUTate and OCEan to do this at runtime.

Also, we are currently working in incorporating the possibility of using machine learning algorithms to infer new behaviour patterns from user behaviour observation. These algorithms could automatically apply the mechanisms to evolve the patterns according to the inferred patterns. However, in this way we would not take into account users' desires since the repeated execution of an action does not imply that users want its automation. Thus, instead of this, the tool will present the inferred patterns to users once a month allowing them to modify or add these patterns if users regard them as appropriated.

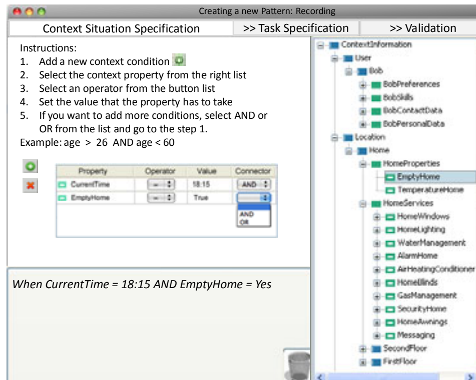



**Fig. 7.** Snapshot of the end-user tool

# 7 Evaluation

In order to evaluate our approach, we carried out two evaluation studies. First of all, we evaluated the feasibility of using models at runtime in our approach. Second, we performed a case study-based evaluation to test our approach in supporting user behaviour automation and evolution. To perform these evaluations, we use a 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 [15], the EMF 2.3, EMFMQ 1.1, EMFMT 1.1 eclipse plugins, Pellet and the OWL API.

## 7.1 System Evolution by Using Models at Runtime

We evaluated the feasibility of using models at runtime in the evolution mechanisms. The model operations that they perform have to be efficient enough so that the system

response is not drastically affected. Thus, we performed an experiment to get the temporal cost of the operations of MUTate and OCEan that access to models. We used our context model and an empty task model to be randomly populated by means of an iterative process. The context model was populated with 100 new individuals 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 the first level each iteration.

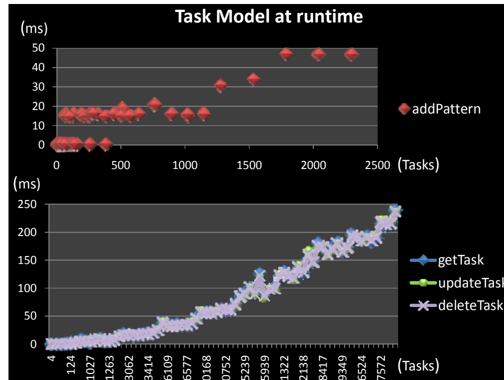



**Fig. 8.** Temporal cost of task model operations

After each iteration, we tested all the model operations of MUTate and OCEan 20 times and calculated the average temporal cost of each one. As an example, the operation over the context model with the highest temporal cost was the *getIndividual* operation, which took 7 milliseconds with 100 individuals and 10 milliseconds with 6000 individuals. This is because this operation has to get the individual by using a SPARQL query, which determines the temporal cost of the operation. Figure 8 shows the temporal cost of the task model operations 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 *getTask*, *updateTask* and *deleteTask* operations. Their costs are very similar since all of them make the same query to get the corresponding task. Even with a model population of 45612 tasks, these model operations provide a fast response (<250 milliseconds). Thus, the response time of using models at runtime is adequate.

### 7.2 A Case Study-Based Evaluation

As a case study-based evaluation of our approach, we developed a prototype to support the automation of the daily tasks that Bob and Sarah perform in their home (examples of these daily tasks have been presented in Section 3). To develop the case study, we specified the behaviour patterns that the system had to automate (with the participation of Bob and Sarah), using the proposed task model. To facilitate the participation of the users, we briefly explained the main concepts of the model and how we modelled the identified behaviour patterns. We found that this model was intuitive

enough for users and expressive enough to model all the identified automation requirements. However, the end-users found it a little difficult to understand some of the used temporal relationships.

To support the functionality needed to execute the system tasks of the patterns, we used the MDD strategy presented in [8] to obtain the code of the required services used to execute the *system tasks*. To evaluate the feasibility of our approach, we ran the system in real-life deployment sessions using *Prosyst*. In the experimental set-up, a scale environment with real devices was used to represent the Smart Home [16].

In addition, we evaluated the usability of the tool for evolving the system automations. To do this, we arranged several sessions in which Bob and Sarah used the tool under our supervision. An observer sat side by side with them and measured (without intervening in their activity) their number of errors and their number of help requests. Since the users had participated in the modelling of the behaviour patterns, it was easy for them to change it. Specifically, 85% of the updates of the patterns were correctly performed; however, they needed particular help to properly establish the temporal relationships between the tasks of a new behaviour patterns. Finally, we measured errors in terms of "wrong clicks" which were, on average, 13% of the overall set of interactions.

## 8 Related Work

Current approaches for automating users' behaviour patterns are predominantly machine learning-based approaches. Some examples are: the MavHome project [2] or the iDorm project [3]. The MavHome project uses prediction algorithms to identify common sequential patterns from data captured from the sensors of a smart home. From this learning, Mavhome builds a Markov model of user behaviour in which patterns are specified through a series of states linked by transitions with certain probabilities. If changes in user behaviour are detected by the algorithms or user feedback, the system is rebooted to obtain an improved Markov model using a new set of observations. The iDorm project models user behaviour by learning fuzzy rules that map sensor state to actuator readings representing inhabitant action. If changes in the user behaviour are detected by the algorithms, rules can be added, modified, and deleted. Thus, the evolution of these models is performed at a low level of abstraction. In addition, these techniques present some problems that can cause the loss of user acceptance of the system, e.g.:

- The algorithms used require **a great amount of training** data and make mistakes during the learning process, causing frustration to users;
- Lack of knowledge about user performed tasks may lead to **automating tasks which the user may not want automation** or reach generalizations in such a way that the automation becomes a burden on the user; e.g., the actions predicted by these algorithms for the *WakingUp* pattern (Section 3) would be: switch on the light, raise the blinds and switch the light off again, instead of directly raising the blinds.

In addition, these approaches could not predict some patterns of our case study because users do not perform these actions (Sarah does not record her favourite program when she is not at home), or they are not performed in the same context (the actions

to be automated when it starts to rain are performed when Sarah and Bob are at home, but not when they are out). In our approach, the evolution is performed at modelling level using concepts of a high level of abstraction, such as task or behaviour pattern, instead of adding states or rules as the above presented approaches. Moreover, users participate in the modelling of the system automation and in its evolution; therefore, the system only automates the tasks that users want automated and without requiring a learning process.

Other approaches try to specify proactive behaviour by using rule-based systems. Some examples are the proposal by García-Herranz et al. [17] and the proposal by Henriksen and Indulska [18]. These approaches are not focused on the automation of whole user behaviour patterns and do not cover the evolution of the rules once the system is running. In contrast, our approach automates whole user behaviour patterns by modelling them using a context-adaptive task model. This model considerably improves the expressivity of the above approaches by using the task concept and the temporal relationships among tasks. Thus, our approach allows users to view the automated actions as a whole task and not as isolated actions that are triggered when some conditions arise.

## 9 Conclusions and Further Work

In this work, we have presented and evaluated a novel approach for confronting the challenge of automating users' behaviour patterns end evolving them at runtime. These patterns are specified in a context-adaptive task model and automated by an engine (MAtE) that interprets the models to execute the patterns as specified. To deal with the evolution of these behaviour patterns, we have implemented MUTate and OCEan that are mechanisms capable of updating these models at runtime [6]. Since these models are interpreted at runtime by MAtE, as soon as they are modified, the changes are applied by the system. In addition, we provide an end-user tool that, by using MUTate and OCEan, allows users to change the patterns by using user-friendly interfaces. Thus, users can update the automated behaviour patterns without having to stop the system using this tool.

As further work, we plan to extend our end-user toolkit to provide interfaces that adapt according to the user preferences, skills and knowledge of the system [20]. According to this information (stored in the context model), the interfaces will provide users with the appropriate end-user techniques to change the automated behaviour patterns.

## References

[1] Neal, D.T., Wood, W.: Automaticity in Situ: The Nature of Habit in Daily Life. In: Bargh, J.A., Gollwitzer, P., Morsella, E. (eds.) Psychology of action: Mechanisms of human action (2007)

[2] Cook, D.J., Youngblood, M., Heierman, I.E.O., Gopalratnam, K., Rao, S., Litvin, A., et al.: MavHome: An agent-based smart home. In: PerCom 2003, pp. 521–524 (2003)

[3] Hagras, H., Callaghan, V., Colley, M., Clarke, G., Pounds-Cornish, A., Duman, H.: Creating an Ambient-Intelligence Environment Using Embedded Agents. IEEE Intelligent Systems 19(6), 12–20 (2004)

[4] Mens, T.: The ERCIM Working Group on Software Evolution: the Past and the Future. In: IWPSE-Evol 2009 (2009)
[5] Bennett, K., Rajlich, V.: Software Maintenance and Evolution: A Roadmap. In: 22nd International Conference on Software Engineering, pp. 75–87 (2000)
[6] Blair, G., Bencomo, N., France, R.B.: Models@run.time. IEEE Computer 42, 22–27 (2009)
[7] OSGI, http://www.osgi.org/
[8] Serral, E., Valderas, P., Pelechano, V.: Towards the Model Driven Development of context-aware pervasive systems. In: Pervasive and Mobile Computing (2009)
[9] Serral, E., Valderas, P., Pelechano, V.: A Model Driven Development Method for developing Context-Aware Pervasive Systems. In: Sandnes, F.E., Zhang, Y., Rong, C., Yang, L.T., Ma, J. (eds.) UIC 2008. LNCS, vol. 5061, pp. 662–676. Springer, Heidelberg (2008)
[10] Eclipse Platform, http://www.eclipse.org
[11] Smith, M.K., Welty, C., McGuinness, D.L.: OWL Web Ontology Language Guide. W3C Recommendation February 10 (2004), http://www.w3.org/TR/owl-guide/
[12] SPARQL Query Language (2008), http://www.w3.org/TR/rdf-sparql-query/
[13] Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Journal of Web Semantics (2007)
[14] Pérez, F., Valderas, P.: Allowing End-users to Actively Participate within the Elicitation of Pervasive System Requirements through Immediate Visualization. In: REV (2009)
[15] Prosyst, http://www.prosyst.com/
[16] EIB, http://www.knx.org/
[17] García-Herranz, M., Haya, P.A., Esquivel, A., Montoro, G., Alamán, X.: Easing the Smart Home: Semi-automatic Adaptation in Perceptive Environments. Journal of Universal Computer Science 14 (2008)
[18] Henricksen, K., Indulska, J.: A Software Engineering Framework for Context-Aware Pervasive Computing. In: PerCom (2004)
[19] Lieberman, H.: Programming by example (introduction). Commun. ACM 43(3), 72–74 (2000)
[20] Pribeanu, C., Limbourg, Q., Vanderdonckt, J.: Task Modelling for Context-Sensitive User Interfaces. In: Johnson, C. (ed.) DSV-IS 2001. LNCS, vol. 2220, pp. 49–68. Springer, Heidelberg (2001)
[21] Shalloway, A., Trott, J.R.: Design Patterns Explained: A New Perspective on Object-Oriented Design. Addison-Wesley, Reading (2004)
[22] Serral, E.: Automating User Behaviour Patterns. Technical Report, PROS - UPV (2009), http://oomethod.dsic.upv.es/labs/media/techreports/TechnicalReport-AutomatingBP.pdf