Serral Asensio, E.; Valderas Aranda, PJ.; Pelechano Ferragud, V. (2013). Addressing the evolution of automated user behaviour patterns by runtime model interpretation. Software and Systems Modeling. doi:10.1007/s10270-013-0371-3.

Addressing the Evolution of Automated User Behaviour Patterns by Runtime Model Interpretation

Estefanía Serral, Pedro Valderas, Vicente Pelechano

Centro de Investigación en Métodos de Producción de Software (ProS)
Universidad Politécnica de Valencia
C/ Camí de Vera S/N, Valencia, 46022, Spain
{eserral, pvalderas, pele}@dsic.upv.es

ABSTRACT

The use of high-level abstraction models not only can facilitate and improve system development but also runtime system evolution. This is the idea of this work, in which behavioural models created at design time are also used at runtime to evolve system behaviour. These behavioural models describe the routine tasks that users want to be automated by the system. However, users' needs may change after system deployment, and the routine tasks automated by the system must evolve to adapt to these changes. To facilitate this evolution, the automation of the specified routine tasks is achieved by directly interpreting the models at runtime. This turns models into the primary means to understand and interact with the system behaviour associated to the routine tasks as well as to execute and modify it. Thus, we provide tools to allow the adaptation of this behaviour by modifying the models at runtime. This means that the system behaviour evolution is performed by using high-level abstractions and avoiding the costs and risks associated to shutting down and restarting the system.

Keywords: System behaviour evolution, routine task automation, models at runtime, runtime interpretation of models.

1. Introduction

The defining characteristic of Model Driven Development (MDD) is that software development's primary focus is models rather than computer programs. Traditionally, the key premise behind MDD has been the use of these models during development activities to improve software development, or even to systematically generate a concrete implementation of the modelled software system. However, the need for fast runtime adaptation required for newer generations of software systems has led a growing number of MDD researchers to also explore the use of models at runtime. Models can provide a richer semantic base than the system code

for runtime decision-making. Thus, the idea is to use models at runtime to represent the part of the system that should be dynamically evolved during system execution.

In this work, we propose the use of models at runtime to evolve the behaviour of Ambient Intelligence (AmI) systems. AmI is a computer paradigm that tries to make real the vision of Mark Weiser [1] where environments are electronically enriched to make them sensitive to user needs. One of the most important goals of building such environments is to serve people in their everyday lives and free them to a large extent from tedious routine tasks. To achieve these goals, AmI systems must automate these routine tasks when needed. Routine tasks, which are also known as user behaviour patterns, are characterized by habitual repetition in similar contexts. Some examples of user behaviour patterns are reactions to things happening around us, such as closing windows and lowering every blind when it starts to rain; others are determined by our lifestyle, such as reading electronic mail and opening certain web pages as soon as we have access to internet; others are determined by our timetable, such as getting up at 8:00 a.m., having a shower, having breakfast and going to work; etc.

Note that many of these behaviour patterns can be known at design time. Thus, we proposed in [2] a context model and a context-adaptive task model in order to describe them at a high level of abstraction. Some of these behaviour patterns might never change in a user's lifetime; however, most of them will. Users' context and circumstances usually change over time and behaviour patterns must evolve to adapt to these changes. Although the proposed models specify the patterns to be automated in such a way that their execution adapts to context, changes in the user behaviour patterns cannot be anticipated at design time; therefore, the evolution of user behaviour pattern is an aspect that must be considered in order to properly automate them.

Considering the environments where AmI systems are deployed (e.g. homes, hospitals, educational centres, etc.), this evolution should be done in a non-traumatic way for users, i.e., without requiring stopping the system to reinstall it. In this work, we face this problem by reusing the proposed models at runtime. We developed a Model-based behaviour pattern Automation Engine (MAtE) [2] that is in charge of automating the behaviour patterns by directly interpreting the models. MAtE uses a context monitor for detecting changes in the system context. Thus, when context changes, MAtE interprets the models at runtime and executes the functionality required to support the user behaviour patterns. Note that this strategy turns models into the only representation of user behaviour patterns. Therefore, if models are evolved at runtime, MAtE immediately reinterprets them making the system behaviour evolve automatically. This is the premise in which the evolution proposed in this paper is based:

The evolution of the system behaviour is achieved by updating the models at runtime.

In the particular case of this work, we focus on evolving the system behaviour that is in charge of automating user behaviour patterns. This is achieved by confronting two of the most important challenges identified in software evolution:

1) Supporting evolution at high levels of abstraction (e.g. design models) [3–5].
2) Supporting post-deployment runtime evolution [6], [7].

It is worth noting that evolution can be handled either automatically by the system itself (it is the system who detects changes in user behaviour and who adapts the automated behaviour patterns to those changes), or manually by humans (the automated behaviour patterns are evolved by humans after analysing the new automation requirements). In this work, we focus on the last possibility due to the fact that an automatic evolution of behaviour patterns in AmI systems may be very intrusive if there is a mismatch between end-users' expectations and system behaviour. Therefore, we present two tools for facilitating the manual evolution of the system: 1) a tool for designers, which allows them to analyse and evolve the patterns by graphically modifying the models; 2) a tool for end-users, which allows them to evolve the automated behaviour patterns by using intuitive graphical interfaces that guide the end-users step by step.

To sum up, the contributions of this paper are the following:

- A novel approach that makes it easier the evolution of the user behaviour patterns that are automated by an AmI system. To achieve this, patterns are automated by interpreting behavioural models at runtime.
- Mechanisms and tools for allowing the runtime evolution of the automated behaviour patterns at a high level of abstraction.

The remainder of the paper is organized as follows. Section 2 introduces the AmI system concept and motivates and characterizes the evolution that we confront in this work. Section 3 presents an overview of our approach. Section 4 introduces the proposed models for specifying behaviour patterns. Section 5 explains how the models are used at runtime to automate the specified behaviour patterns. Section 6 describes the provided high level mechanisms for evolving the automated behaviour patterns at runtime. Section 7 presents the tools developed for allowing the AmI system behaviour to be evolved at a high level of abstraction after deployment. Section 8 evaluates the presented approach. Section 9 discusses the related work. Section 10 presents the most important benefits of the proposal. Finally, Section 11 explains the conclusions and the further work.

2. Motivation: Evolution of User Behaviour Patterns

AmI systems are focused on providing smart environments to users. According to Weiser [1], a smart environment is a “physical world that is richly and invisibly interwoven with sensors, actuators, displays, and computational elements, embedded seamlessly in the everyday objects of our lives, and connected through a continuous network". In this context, several terms are used in the published literature for talking about similar concepts. The main differences depend on the context of use: Academy vs Industry and USA vs Europe. While researchers in the United States were working on the vision of *ubiquitous computing*, the industry, leaded by IBM, coined the term *pervasive computing*. At the same time, the European Union began promoting a similar vision for its research and development agenda through the notion of Ambient Intelligence. Although subtle differentiations could be done between these terms according to their etymological meanings (neither ubiquitous implies intelligence, nor intelligence implies pervasiveness, etc.), we can state, in general terms, that the main idea or vision behind them is the same. Thus, in this work are considered equivalent concepts.

In a nutshell, AmI systems’ main goal is to control devices in order to provide users with complex behaviours. To achieve this, an AmI system provides a set of services. There are services in charge of interacting with devices to provide the environment with an active behaviour (such as those to turn lights on, raise blinds, play a movie on a television, or activate an alarm); and there are other services in charge of interacting with devices to sense the environment state (such as those to get the light intensity, to detect people’s presence, or to obtain climatological information).

It is important to note that in AmI, services are not considered in an isolated manner, but in conjunction with the rest of services provided to control the existing devices. In our approach, a behaviour pattern can be seen as a specific coordination of these services that starts when specific conditions are fulfilled in order to provide the user with an automated behaviour. Thus, each user behaviour pattern that must be supported by the system is designed as a coordination of services’ execution. Figure 1 represents this notion in a graphical way.

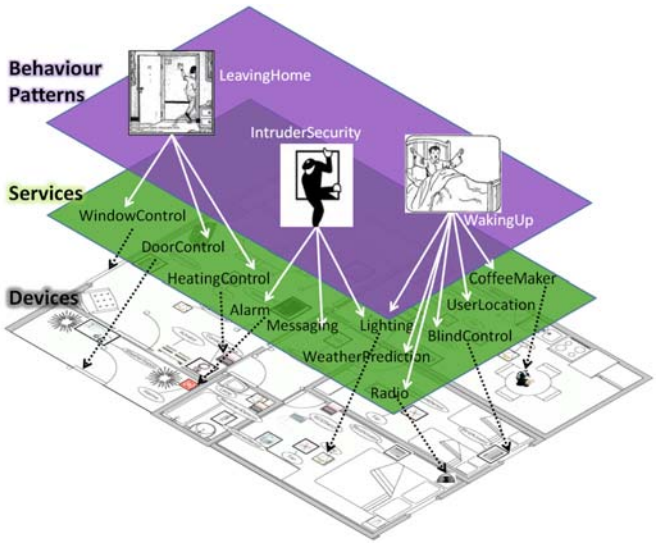


Figure 1 AmI system architecture

As a motivation example, we present a set of AmI services that are in charge of interacting with different objects of a Smart Home. In order to better understand them, we avoid presenting technology implementations and show the services from a conceptual point of view, by means of a UML class diagram, as shown in Figure 2. In this example, we consider pervasive services as classes with operations that allow interaction with the physical environment. Note, however, that other strategies such as considering them as web services or distributed components could be used.

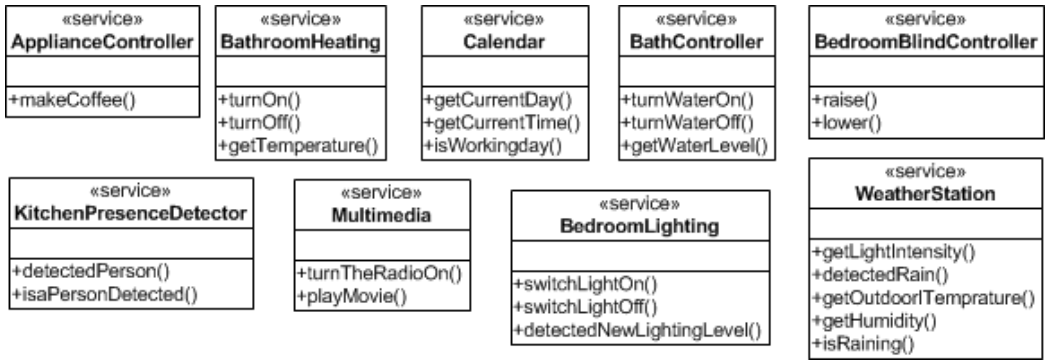


Figure 2 Example of pervasive services

The service operations presented in Figure 2 allow us to individually interact with different objects, for instance: we can turn the radio on and off; switch the light of the bedroom on and off; or know if a person has been detected in the kitchen. However, theses service operations must be executed in a coordinated way in order to support user behaviour patterns.

Figure 3 shows an example of a pseudo-code representation of the execution of a behaviour pattern that supports the waking up of a user. As shown, 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 turns the radio on; 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.

```
when Calendar.getCurrentTime()==7:50 and Calendar.isWorkingDay():
    if BathroomHeating.getTemperature()<28 then
            BathroomHeating.turnOn();
    wait(10);
    Multimedia.turnTheRadioOn();
    if WeatherStation.isSunny() then
            BedroomBlindController.raise();
    else BedroomLighting.switchOn();
    when KitchenPresentDetector.isaPersonDetected():
        CoffeeMaker.makecoffee();
```

Figure 3 Example of behaviour pattern in pseudo-code

Let's suppose now that the user's preferences change after deployment and s/he prefers orange juice for breakfast rather than coffee; or users prefer to raise blinds when it is not raining rather than only on sunny days; or users prefers to apply this behaviour pattern on weekends rather than on working days. These simple changes require stopping and redeploying the system if ad-hoc solutions, such as the shown in Figure 3, are used. To avoid this, we require engineering solutions that allow us to perform this type of post-deployment evolution without stopping the system.

2.1. Evolution Characterization

In this section, we characterize the evolution supported in this work by following the taxonomies published in [8][9].

Lientz and Swanson classify software evolution by answering to the question ***why***. They describe three intentions for software evolution: to perfect the system (perfective), to adapt the system (adaptive) and to correct the system (corrective). Buckley et al. complete the taxonomy of Lientz and Swanson by answering the questions: *when*, *where*, *what*, *how*, and *who*. The ***when*** dimension characterizes evolution from two main aspects: (1) the phase of the software life-cycle of which it is performed, which delimitates three types of evolution: at compile-time (static), at load time, and at run-time (dynamic); and (2) the anticipation of the required evolution, which delimitates two types of evolution: anticipated, if evolution can be foreseen at design time; and unanticipated, if evolution needs arise from using the system. The ***where*** dimension characterizes the software artefacts where changes are made (requirements, architecture, design, source code, documentation or test suites). The ***what*** dimension characterizes evolution from the point of view of the system attributes that may have conditioned it. These attributes properties can be *availability* (whether the software system has to be permanently available or not), *activeness* (evolution is either reactive if system changes

must be driven by an external agent or proactive if the system is able to self-change by using the information received from monitors), *openness* (whether or not the system must allow for extensions), and *safety* (whether or not safety aspects must be considered at compile time and/or at runtime). The ***how*** dimension characterizes evolution from the point of view of the degree of automation and formalism that is introduced in the mechanisms provided to support it. Finally, although the ***who*** dimension does not have a concrete taxonomy due to its great variety, it delimitates the stakeholders involved in software evolution.

In this work, we propose designing user behaviour patterns in models, and interpreting the models at runtime. Thus, user behaviour patterns are evolved by evolving the models. Taking into account the above-introduced dimensions, this evolution is characterized as follows:

- Why: user behaviour patterns need to be evolved to adapt to new users' needs and circumstances. Thus, this evolution has to be adaptive.
- When: it is dynamic and unanticipated. It is dynamic because user behaviour patterns should be evolved at runtime, without stopping the system. It is unanticipated because the way in which user behaviour patterns must change cannot be foreseen at design time. These changes can be produced by modifications in users' habits due to personal reasons, bad experiences with the system behaviour, unplanned alterations of the physical environment, and so on.
- Where: the only runtime artefacts that represent user behaviour patterns are the models. Thus, models are where changes are made to evolve the user behaviour patterns.
- What: AmI systems must be always available in order to evolve user behaviour patterns. Note that users may need it at any time. In addition, user behaviour patterns should be evolved in a proactive way. Note that AmI systems can be self-adapted by analysing user behaviour and predicting a required evolution. However, it is desirable to confirm the predicted evolution with users before applying it in order to not result intrusive [10]. AmI systems can be considered as open systems because they are developed in such a way that new services can be plugged-in to support new devices. Finally, user behaviour evolution must provide behavioural safety in order to avoid a system behaviour that is not desired by users.
- How: the mechanisms proposed to evolve the system are the models. Models are formalized by meta-models. Thus, the evolution is driven by the restrictions defined in the meta-models. In this sense, the mechanisms to perform evolution present a high degree of formalism. User behaviour patterns must be evolved on demand, when users require. Thus, evolution is not automated.

- Who: the evolution of the automated behaviour patterns should be driven by developers and by the end-users themselves. Note that some AmI environments such as hospitals or educational centres require a flexible evolution that can be done in a non-traumatic way by the end-users.

3. Approach Overview

In this section, we introduce an overview of how behaviour pattern automations are evolved at runtime by using our approach.

First of all, we have to consider that behaviour patterns are described in a task model and a context model (both presented in the next section). These models are interpreted at runtime by a Model-based behaviour pattern Automation Engine (MAtE), which executes the corresponding behaviour patterns according to the current context (see Figure 4). When a context changes (e.g. a user enters a room), MAtE checks whether there is any pattern that has to be executed. If so, MAtE retrieves the pattern instance from the models and executes the corresponding pervasive services as specified in the pattern, taking into account the context conditions defined over properties of the context model.

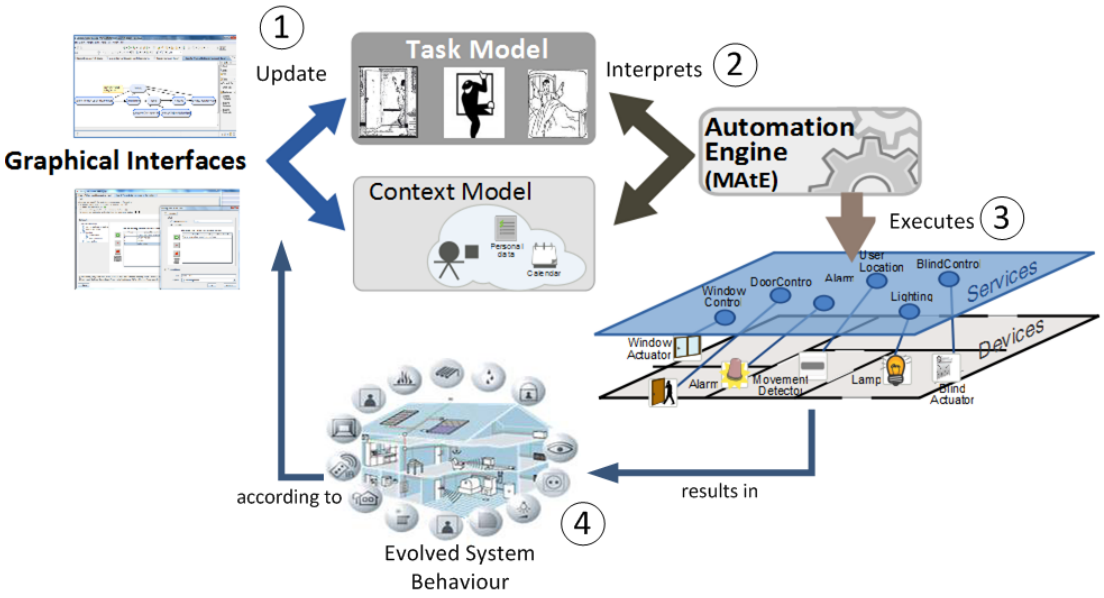


Figure 4. Evolution overview

Thus, the evolution of automations can be done by just updating the models. This evolution is applied next time the modified information is interpreted from the models, resulting in the evolved system behaviour. For instance, if a new behaviour pattern is added, it is executed the next time that a context change triggers the execution of the pattern; if a behaviour pattern is modified, next time it is executed, the changes are considered; if a behaviour pattern is deleted, it is not executed anymore.

In order to update models at runtime we provide two different tools aimed to be used by designers and end-users. These tools abstract the Java implementations required to update models providing interfaces that allow evolving pattern automations in a graphical way. Thus, the process followed in order to evolve the behaviour patterns is as follows:

1. Designers or end-users use the provided graphical tools to update the models. With these tools, behaviour patterns can be added, updated or deleted from models.
2. When context changes, MAtE reinterprets the models (considering the changes made in the previous step) and loads the patterns that must be executed.
3. MAtE searches for the pervasive services that must be executed to carry out each pattern, and executes them according to the pattern descriptions.
4. Pervasive services are executed considering the changes performed in Step 1. Thus, modifications in models result in the evolution of the system behaviour at runtime, without stopping the system.

It is important to note that when a behaviour pattern is being executed, its current execution cannot be modified; the changes performed to a pattern will be applied next time the pattern has to be executed.

Although this work is focused on AmI environments, note that the approach proposed to evolve system behaviour can be generalized to other domains. To do so, we need models that describe the behaviour to be evolved, tools for editing them, an engine that interprets the models, and mechanisms that allow the engine to look up the functionality that supports the behaviour described in models.

Finally, it is worth remarking that the contribution of the presented work is the adaptation of system behaviour by directly evolving the models at runtime. We combine the research areas of models at runtime, user task automation and AmI systems to build an approach for automating behaviour patterns by interpreting at runtime the models where the patterns are specified. As Section 5 will show, this approach makes easier the evolution of the behaviour patterns at runtime and at a high level of abstraction: since MAtE interprets the models and executes the routines as specified, the routines can be evolved by directly modifying the models at runtime. To facilitate this evolution, we present, in Section 6 and 7, appropriate mechanisms and tools for developers, designers and end-users.

4. Modelling User Behaviour Patterns

To specify the user behaviour patterns that users want to be automated, we proposed a Doman Specific Modelling Language (DSML) composed by two models: a context model, to specify the

context on which the behaviour patterns depend, and a context-adaptive task model, to specify the behaviour patterns according to the context described in the context model.

4.1 The Context Model

A suitable model for handling, sharing and storing context is essential for automating user behaviour patterns in an unobtrusive manner. Different context models have been proposed to capture context in Pervasive Computing. Some of the most important examples are: object-oriented models such as the ones proposed by the projects CORTEX [11] and Hydrogen [12]; key-value models such as the one used by Dey in the Context Toolkit [13]; graphical models such as ContextUML [14] and CML [15]; etc. However, several studies [16–18] state that the use of ontologies to model context is one of the best choices. Therefore, we use the context ontological model proposed in [2] to semantically describe the required context. The concepts, and their main attributes and relationships provided by this model are shown in Figure 5. This model allows representing *users*, l*ocations*, e*nviroment properties*, *policies*, t*emporal properties, system services* and e*vents* (which could be, for instance, context changes or task execution).

The model is specified in the Ontology Web Language (OWL) [19]. OWL is an ontology markup language that greatly facilitates knowledge automated reasoning and is a W3C standard. Thus, the classes of the ontology are defined as OWL classes, the attributes as data properties and the relationships as object properties. The context specific to the system is defined as OWL individuals, which are instances of these classes. Figure 5 shows also some context instances of the example explained in Section 2. For instance, Bob is defined as an individual of the *User* class, Kitchen is defined as an individual of the *Location* class, Temperature is defined as an individual of the *EnviromentProperty* class, etc.

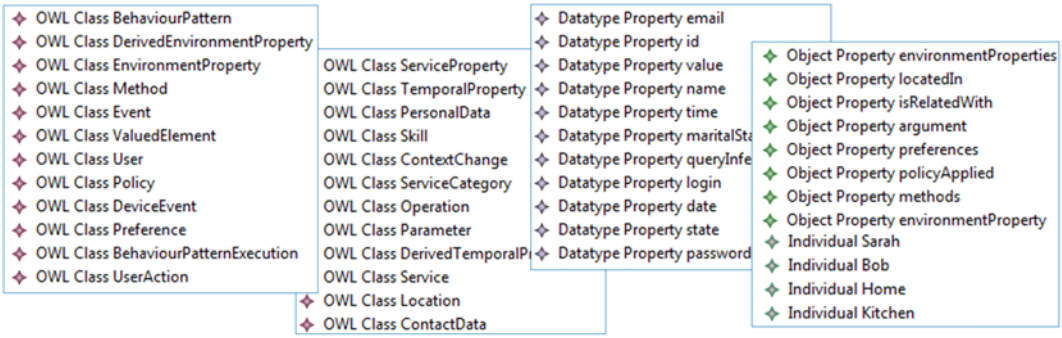


Figure 5 A context model example

To easily create the context model, we used the EODM plugin[1] of the Eclipse platform, which allows an OWL model to be visualized and edited using a tree graphic editor (as shown in Figure 5), as well as using OWL code. **4.2 The Context-adaptive Task Model**

The proposed context-adaptive task model is based on the *Hierarchical Task Analysis (HTA) technique* [20], which breaks down tasks hierarchically into other tasks. We chose a HTA-based model instead of other task models because it provides us with more facilities to describe user behaviour patterns [2]. In particular, we proposed defining a task hierarchy for each behaviour pattern. The root task (depicted by an ellipse, see Figure 6) represents the behaviour pattern and has an associated context situation (depicted with a note associated to the root task), which defines the context conditions whose fulfilment enables the execution of the behaviour pattern. This root task can be broken down into *composite tasks (*which are intermediate tasks*)* and/or *system tasks* (which are leaf tasks). Composite tasks (depicted by ellipses, see Figure 6) are used for grouping subtasks that share a common behaviour or goal. System tasks are depicted by ellipses with a blue border and are associated to pervasive services that are in charge of implementing the functionality that perform them. These services can be developed by using any development method; the only restriction is that they must be implemented in Java/OSGi[2] in order to be used by the infrastructure presented in the following sections. Additional information about service implementation can be found in [21].

Both composite and system tasks can have a context precondition, which is depicted between brackets within the task ellipse (see Figure 6, tasks *turn on bathroom heating*, *raise bedroom blinds* and *switch bedroom lights on*). This precondition defines the context situation that must be fulfilled so that a task is performed (if the precondition is not fulfilled, the task will not be executed). Child tasks inherit the context preconditions of their parent task.

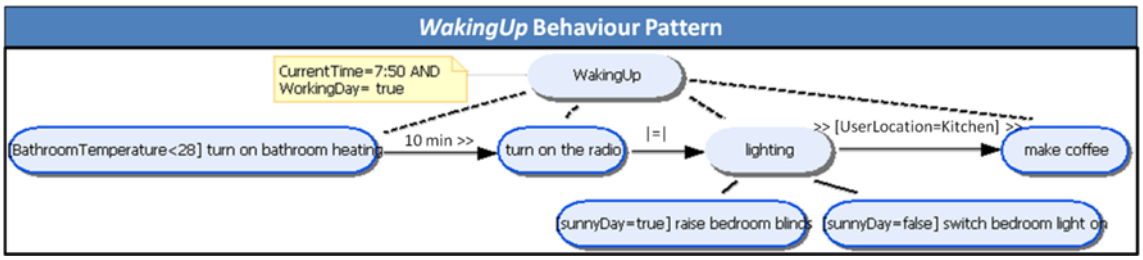


Figure 6 An example of behaviour pattern modelling

In order to break down a behaviour pattern or a composite task into simpler tasks, we propose two task refinements: the exclusive refinement (represented by a solid line) and the temporal

[1] EODM plugin - http://www.eclipse.org/modeling/mdt/eodm/docs/articles/EODM_Documentation/

[2] Open Service Gateway Initiative (OSGi) - www.osgi.org

refinement (represented by a broken line). The exclusive refinement decomposes a task into a set of subtasks in such a way that only one subtask will be executed (disabling the others). The temporal refinement also decomposes a task into a set of subtasks; however, this refinement provides constraints for ordering the execution of these subtasks. These constraints are depicted by means of arrows between the related tasks. A complete description of these operators can be found in [2].

As example, Figure 6 shows the modelling of the *WakingUp* pattern[3]. It is initiated at 7:50 a.m. on a working day (see context situation associated to the root task). The root task is refined into four tasks by temporal refinements in such a way that they must be executed following the specified constraints. According to the first task (the one situated at the left side), the system starts by *turn*ing *on the bathroom heating*. Ten minutes later (see temporal association between both tasks), the system must *turn on the radio*. This second task is related with the *lighting* task by means a concurrent task temporal operator (|=|), which indicates the execution order of the *turn on the radio* and *lighting* tasks is not relevant.

The Lighting task is a composite task that is refined in two tasks (*raise bedroom blinds* and *switch bedroom light on*) by exclusive refinements in such a way that only one of these tasks will be executed (depending on whether it is a sunny day or not). After the *lighting* task is executed, the system completes the behaviour pattern by *mak*ing *coffee* when the user is in the kitchen (see temporal restriction between tasks).

To conclude this section, we formalize the elements that can appear in the task model (i.e., its abstract syntax) by defining its metamodel, which is shown in Figure 7. The main element of this metamodel is the Task class, which can be System or Composite. The composite tasks can be refined using a Temporal Refinement or an Exclusive refinement into more specific tasks until they can be considered System tasks (i.e., executable tasks), which are related to a service that can execute it. This service is specified by its identifier and the name of the method that should be specifically executed for carrying out the task. When a composite task has been refined by using the temporal refinement, its subtasks are related by means of a Temporal Relationship. A Behaviour Pattern is defined as a subtype of composite task. Each Behaviour Pattern has to be related with a Context Situation element, which is a Context Condition that must be fulfilled to activate the behaviour pattern. Temporal relationships can also be related to a context condition. In the same way, tasks can also have a Context Precondition, which is a subtype of Context Condition. Additional information about the model and its metamodel can be found in [2].

[3] This pattern is used as a running example along the rest of the paper. Other different examples of behaviour patterns can be found in http://www.pros.upv.es/art/

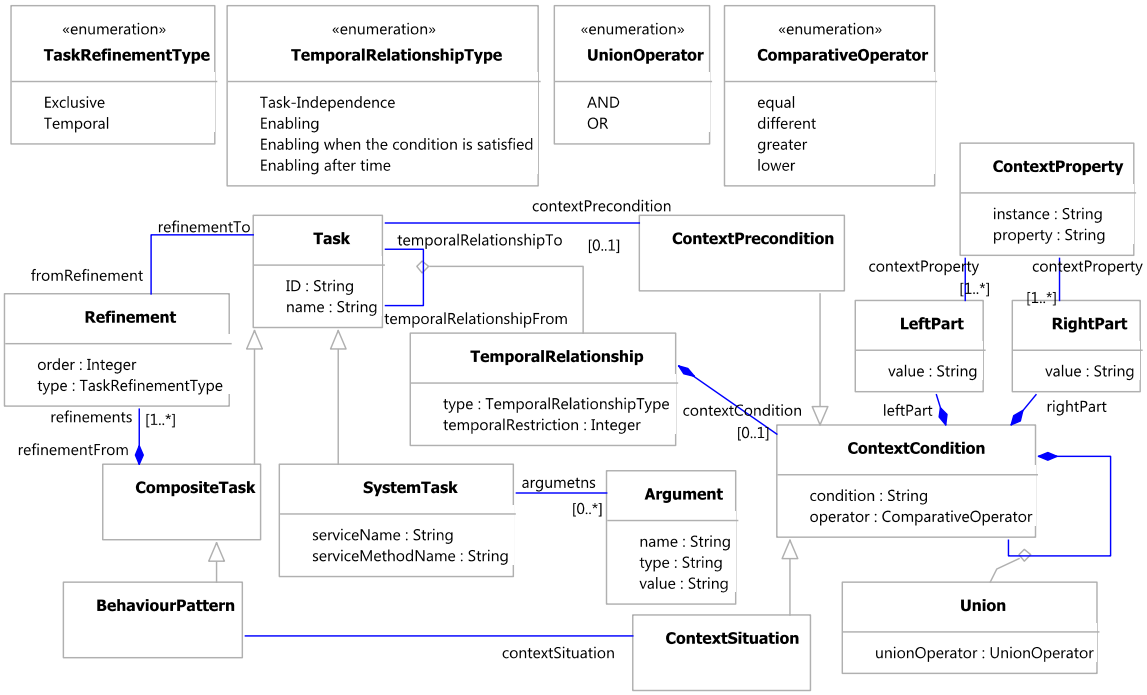


Figure 7 Overview of the task model metamodel

5. Automating User Behaviour Patterns through Model Interpretation

In this section, we present a software infrastructure that interprets the above presented models at runtime in order to automate the specified behaviour patterns. To do this, first, we justify why model interpretation has been chosen; next, the developed software infrastructure is explained discussing the architectural decisions taken; finally, we introduce an overview of some implementation issues.

5.1 Why Model Interpretation?

The models presented in the previous section describe the behaviour patterns that have to be automated in order to satisfy users' needs. These models are not only abstract representations of the behaviour patterns, but they are also machine-processable artefacts with enough precision to be used as executable models [2], [22]. In order to turn executable models into a reality two strategies can be applied: code generation, in which models are executed by means of translating them into an equivalent system code (such as in [23], [24]), and model interpretation, in which models are executed by using an interpreter/engine that directly executes what is specified (such as in [25], [26]).

Code generation is usually better, for instance, when the use of system resources is critical for the system: code generation usually needs less system memory (in model interpretation models are managed in memory, which produces a higher usage of it), and provides a better system

response time (in model interpretation, the interpreter needs to analyse the models, which may impact on the response time).

However, the objective of this work is automating behaviour patterns in smart homes in such a way that the patterns can be evolved after system deployment, even by end-users. In this context, the use of system resources is not critical: system performance and memory usage are not a problem for the size of the models managed in this domain (see Section 8.2 for more detailed). In addition, with a focus on facilitating this evolution at runtime, model interpretation provides more benefits than code generation [28]. These benefits are the following ones [27]:

- It enables faster changes: changes in the model do not require an explicit regeneration, rebuild, retest, and redeploy step. This will lead to a significant shortening of the turnaround time.
- It enables changes at runtime: because the model is available at runtime it is even possible to change the model without stopping the part of the system that provides the functionality that is changed.
- It is easier to update and scale: it is easier to change the interpreter and restart it with the same model. It is not needed to generate the code again using the updated generator. The same can hold for scaling: scaling an application means initializing more instances of the interpreter executing the same model, in this way, parallel behaviour could be performed. For instance, parallel instances of MAtE may be needed to be applied in health care applications, where many behaviour patterns should be performed at the same time, context is changing quickly and the time response is critic.
- It is more secure: The model interpreter provides an additional layer of security when modifications should be done in aspects captured by models. When applications are generated from models they are deployed in a specific system and modifications may require accessing the file system or other system resources. When models are interpreted, modifications are made by updating models and the rest of the system is abstracted by the interpreter. In this way, the interpreter provides an additional layer on top of the infrastructure and everything underneath is abstracted away. This also makes it possible that the automation infrastructure (OSGi environment, interpreter and pervasive services) is deployed in a different system than the models, similar to the idea of Platform as a service (PaaS). Thus, any user could have a pervasive system that automates his/her behaviour patterns by deploying his/her models (i.e., context-adaptive task model and context model) in the cloud, without the cost of developing the software required to automate the patterns.

For these reasons, we have applied a model interpretation strategy. The automation of behaviour patterns is carried out by an engine that directly interprets the models at runtime. This allows the models to become into the only representation of the automated behaviour patterns, which enables that the automated behaviour patterns are evolved by directly updating the models at runtime.

Finally, it is important to note, that at the end, the use of one technique or another will depend on the system domain and its requirements. We have chosen model interpretation because it provides us with more benefits for dealing with evolution in smart home systems..

5.2 Software Infrastructure

AmI systems are deployed into active environments whose context properties are continuously changing. These properties constitute the context in which the system is executed and become a key element to decide when to execute one behaviour pattern or another. Thus, to carry out the behaviour patterns as specified in the models, two important things have to be considered: 1) the opportune context in which each behaviour pattern has to be triggered; and 2) the tasks to be executed in each behaviour pattern taking into account their execution order and the context in which they have to be executed.

The context properties in which each behaviour pattern has to be triggered are specified in the task model as a context situation related to each behaviour pattern (see Section 4.2). Thus, in order to automate a behaviour pattern as specified in the models, the following steps must be performed:

1. Detect if the context properties specified in the context model change. To do this, context should be continuously monitored.
2. When context changes (i.e., a sensor detects a change or a service is executed), the context model must be properly updated.
3. The context-adaptive task model must be interpreted to detect if the new context satisfies the context situation of some behaviour pattern.
4. If the context situation of a behaviour pattern is fulfilled, the behaviour pattern must be performed. To do this, the corresponding behaviour pattern instance stored in the context-adaptive task model must be retrieved. The retrieved information must be interpreted and the appropriate system tasks of the pattern must be executed according to its specification and the current context stored in the context model.

Therefore, considering the service-orientation of AmI system architecture (see Section 2), where a compound of services is provided to control the devices that exist in the environment, we

provide two additional architectural elements in order to automate user behaviour patterns through model interpretation (see Figure 8):

1. **A context monitor**, which is in charge of monitoring the execution of services in order to capture context changes (step 1). When a change in context is detected, the monitor adds the corresponding *Event* instance to the context model and updates its corresponding context properties at runtime (step 2). Once the context model has been updated, the context monitor informs the engine described below about this change (step 3).

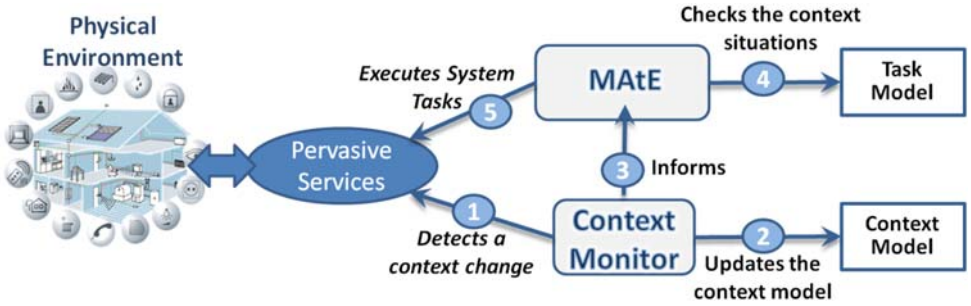


Figure 8: Architectural elements of the software infrastructure

2. **Model-based Automation Engine (MAtE)**, which is in charge of executing the specified behaviour patterns in the appropriate context. To do this, when the context monitor informs MAtE about a context change, MAtE checks the context situations specified in the task model (step 4). If the context situation of a behaviour pattern is fulfilled, MAtE automatically executes that pattern (step 5). To do this, MAtE executes the appropriate system tasks of the behaviour pattern by taking into account its tasks' relationships, its refinements, and the current context. To execute a system task, MAtE executes the pervasive service that is associated to the task.

Thus, the architecture of our proposal is made up of the following elements: (1) the physical environment in which a set of devices (sensors and actuators) are installed; (2) the services that are in charge of controlling these devices; (3) the context and context-adaptive task models that are in charge of describing how the AmI services should the automated in order to satisfy user's needs; (4) a context monitor, which is in charge of monitoring the service execution in order to detect changes in the environment properties, and update the context model accordingly; and (5) the MAtE engine, which is in charge of interpreting the context-adaptive task model in order to execute AmI system's services according to the desired user behaviour patterns.

5.3 Implementation Issues

Both the Context Monitor and MAtE have been developed using Java/OSGi technology. We have chosen OSGi, because it is widely used for developing AmI systems due to the numerous

important benefits that it presents: (1) it provides a service-oriented architecture; (2) it allows device discovery using low level protocols such as EIB, Lonworks or UPnP; (3) it provides dynamic bundle loading and updating, and service lookup; (4) it acts as a bridge to the final device drivers in order to provide a software representation of devices; (5) there is a dynamic coupling of services and devices guaranteeing a high level of robustness; etc.

The OSGi framework allows registering services by publishing their interfaces using the framework's service registry. This registration makes the services discoverable through the registry so that a certain service can be searched when needed. The framework also manages dependencies among services to facilitate coordination among them. These dependencies are implemented using Wire objects. A Wire object acts like a communication canal 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. To achieve this, the Producer service must implement the OSGi Producer interface, while the Consumer service must implement the OSGi Consumer interface. Wire objects implement methods to store and manage the interchanged values. The Consumer interface contains methods to add Wire objects to the consumer and to retrieve them, while the Producer interface contains methods to add Wire objects to the producer, to update the Wire values, and to notify changes to consumers.

Thus, the components of the software infrastructure have been connected as follows (see Figure 9): MAtE and the context monitor are developed as Java OSGI services. They are connected by an OSGi wire. In this wire, the context monitor plays the role of producer, because it informs MAtE about context changes, while MAtE plays the role of consumer, because it needs to know the context changes detected by the context monitor. The context monitor is also connected with each one of the services of the AmI system by a wire. In these wires, the services act as producers because they provide the context monitor with information about context, while the context monitor acts as a consumer, because it uses the information produced by the services to update the context model and notify MAtE. In addition, all services provided by the AmI system are published in the OSGi registry so that MAtE can discover them when they must be executed. To execute a service, MAtE uses the Java reflection capabilities.

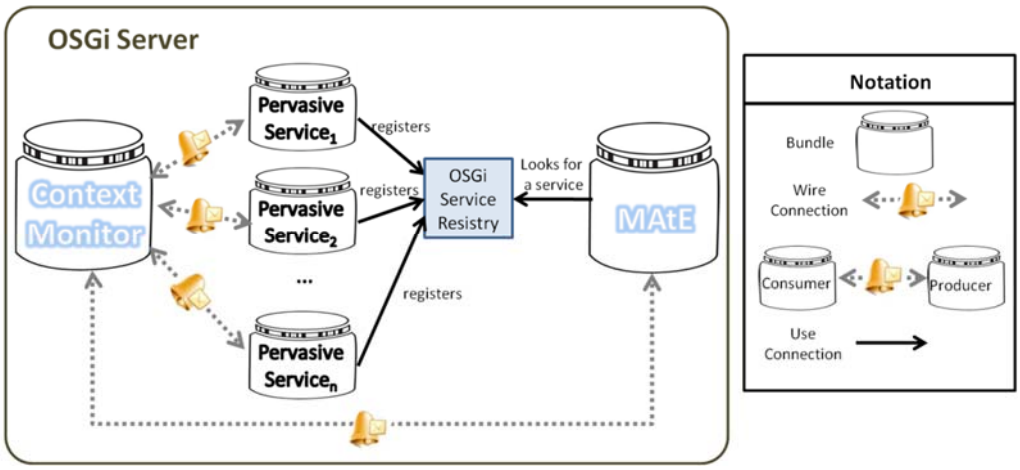


Figure 9. The software infrastructure in the OSGi server

Thus, when context changes, a service detects it and notifies the Context Monitor. The Context Monitor updates the context model and notifies MAtE about the context changes. When this happens, MAtE interprets the context-adaptive task model to check if some context situation is fulfilled and carry out the behaviour patterns whose context situation is fulfilled. To carry out a behaviour pattern, MAtE searches for the services that need to be executed using the service registry, and executes them as specified in the task model.

6. High-level Mechanisms for Evolving the Behaviour Patterns

As said, the models are the unique representation of the automated behaviour patterns. These models are directly interpreted at runtime to execute the behaviour patterns as specified. For this reason, when the models are changed, next time they are interpreted by MAtE, the changes will be applied. Thus, in order to allow developers to evolve the automated behaviour patterns, we provide them with high-level mechanisms to manage the models at runtime. We next present these mechanisms and analyse the different types of evolution that can be performed using them.

6.1 Mechanisms for Managing the Models at Runtime

We designed and developed the following mechanisms for managing the models at runtime: OCean, which allows the context model to be updated at runtime; MUTate, which allows the task model to be updated at runtime; and a concurrency module, which ensures that the evolutions are performed in a consistent manner. OCean and MUTate are developed as java APIs so that they provide the same vocabulary defined in the ontology and the task model metamodel respectively. This facilitates that they can be used by third party systems in any software platform. These APIs provide the same high-level abstraction concepts used in the modelling language, achieving that the models can be managed using the same concepts. In addition, these APIs ensure that the changes are syntactically correct because they have to be in

accordance with the context ontology and the task metamodel definition. To develop these APIs we have applied the best practices on API development recommended in the literature [28], [29].

Ontology-based Context model management mechanisms: OCean

OCean allows the management of the context model respecting the vocabulary established in the context ontology. Since context is captured in the context model as OWL individuals, the OCean API allows creation, retrieval, modification, and deletion of any individual of the context model. To achieve this, the API provides a Factory class for creating new individuals in the context model and getting those that have already been created. Also, the API provides an implementation class (and its corresponding Java interface) for each one of the OWL classes defined in the context ontology. Each class allows its individuals to be obtained, modified, and deleted. To achieve this, each of these classes provides:

- An attribute for each one of the properties and relationships of its OWL class; e.g., the *Person* OWL class has *Name* and *preference* as attributes.
- Get and set methods for each one of these attributes; e.g., *getPreference, setName.*
- Add and remove methods for the attributes whose type is a List. These methods allow an element to be directly added to the list; they also allow one to remove all the elements of the list; e.g., *addPreferences, removePreferences* method.

To generate this API from the context ontology we have used the Jastor[4] tool. We have extended this generated API with a *Model* class that allows a context model to be opened and saved. To make easier the update of the context model according to context changes, this class also provides a more generic API that allows us to manage the individuals of the ontology independently of its class. Specifically, this *Model* class provides methods such as *setProperty* or *getProperty* to update and obtain a property of any individual; or *addRelatedInstance*, to relate an individual to another individual. Furthermore, this Model class provides facilities for querying the model using SPARQL[5] , which is a graph-matching query language recommended by the W3C that allows queries to be built to search for certain individuals in the context model. Specifically, the class provides the method *checkCondition* to check whether a context condition is fulfilled or not.

To implement this class, we used Jena 2.4[6], the OWL API 2.1.1[7], and the Pellet reasoner 1.5.2. [30]. Jena is a Java framework for building Semantic Web applications that provides a programmatic environment for OWL and SPARQL and includes a rule-based inference engine.

[4] http://jastor.sourceforge.net/
[5] http://www.w3.org/TR/rdf-sparql-query/
[6] http://jena.sourceforge.net/
[7] http://owlapi.sourceforge.net/

We used Jena to open the OWL model and save the performed changes in it. The OWL API is an open-source API that provides facilities for creating, examining and modifying an OWL ontology. We have used the OWL API to access and modify the individuals of the context model. Pellet is an open-source tool that provides reasoning services for OWL ontologies. Pellet facilitates accessing the information stored in the ontology and allows us to launch a SPARQL query against the context model using Jena.

Model-Based User Task management mechanisms: MUTate

Similar to OCean, MUTate must allow the management of the task model respecting the vocabulary established in the task model metamodel.

In particular, we have used the Eclipse Modelling Framework (EMF) plugin of the Eclipse Platform, which provides us with many benefits for managing an XMI model at runtime. From the metamodel of the task model in Ecore (see Figure 7), we use EMF to generate the MUTate API for managing a task model. This API provides a Factory class for creating new instances of the task model metamodel elements and getting those that have already been created. In addition, MUTate provides a Java interface and an implementation class for managing the instances of each one of the classes of the metamodel (e.g. *BehaviourPattern* class, *Task* class, *TemporalRelationship* class, etc. See Figure 7). Each of these classes 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 *subTasks* as attributes.
- Get and set 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., *addSubTask* method.

Some of these Java classes represent context conditions, such *ContextSituation* or *ContextPrecondition*. These classes provide the *checkCondition* method as well, which queries the context model to check the corresponding context condition by using OCean.

In order to get additional information about these two APIs, their complete specification can be downloaded from http://www.pros.upv.es/art/.

Concurrency Management: MUTate and OCean can be used for managing the models at runtime. However, to do this at the same time that MAtE and the context monitor interpret the models, concurrency mechanisms are needed for maintaining the consistency of the models. For instance, at the same time that we are evolving the models, MAtE can be interpreting the task model to automate some behaviour pattern (or the Context Monitor can be updating the context model). Therefore, during partial modifications, MAtE or the Context Monitor could access an inconsistent model. Note that one of the objectives of this work is to evolve the configured

automations without stopping the system. Thus, in order to deal with this problem, we have integrated MUTate and OCean with a module that controls the access to models by following a strategy typically used in the management of concurrency in databases: Multiversion Concurrency Control (MCC) [31]. The main advantage of using the MCC model of concurrency control is that reading operations or reading and writing operations do not lock among them.

Basically, the MCC strategy proposes the use of multiple versions of the models, although only one is considered the master copy. As shown in Figure 10, when the models have to be updated, the concurrency module makes a dirty copy where the changes will be made. When all the changes are finished, the concurrency module updates the master copy accordingly. This is done in a transactional environment in order to guarantee the isolation of this action with respect others. To do this, we use the transaction facilities provided by EMF for the task model and the transaction facilities provided by Jena for the context model. Thus, we achieve that the models that are interpreted are always consistent.

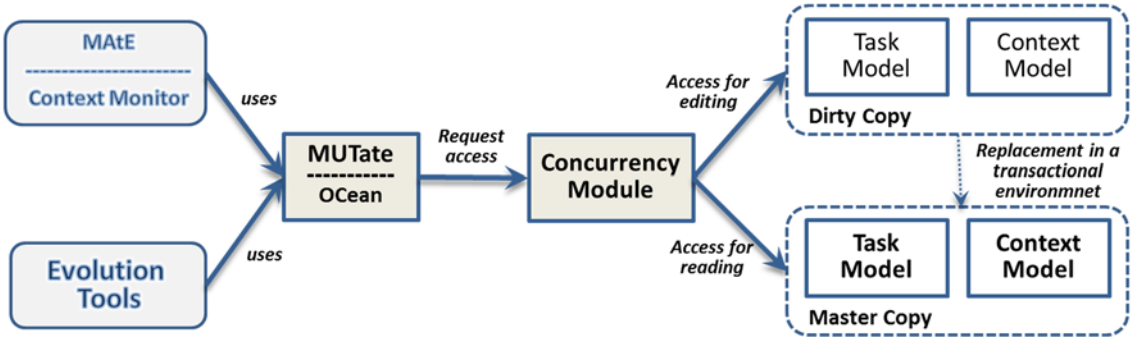


Figure 10. Model evolution architecture

6.2 Supported Evolution

Using these high-level mechanisms, any change that respects the task metamodel and context ontology syntaxes can be performed to evolve the automated user behaviour patterns. These mechanisms allow new behaviour patterns to be created, and also to modify or delete those that are already specified. According to the model interpretation process (see Section 5.2.), when a behaviour pattern is changed, these changes are applied the next time it has to be executed.

Next, we show how these mechanisms can be directly used to perform different types of evolution. These mechanisms can be also used to build graphical tools that facilitate to perform these evolutions. In Section 7 we will explain two different tools and show how the same evolutions can be performed by using them.

In order to show the effects that evolving the context-adaptive task model has in the runtime system behaviour, we show a trace of the executed services. To do that, we use the WakingUp user behaviour pattern presented along the paper. In order to simulate a real interaction with sensors, we have used a device infrastructure made up for simulation purposes. This

infrastructure contains several devices such as presence detectors, light intensity sensors, weather stations, blind controllers, and so on. This infrastructure is shown in Figure 11. In order to simulate the interaction with devices that are difficult to have in an academic environment (e.g. a smart coffee maker) we have used a device simulator. This simulator was presented in [24] and allows developers to define virtual devices and control them using an intuitive user interface.

Figure 11 Device infrastructure with simulation purposed

Evolving the executed services. New tasks may be required to be automated in a behaviour pattern, other tasks may not be needed anymore, and some tasks may need to be slightly modified. For instance, in the *WakingUp* pattern, the user may want that the system informs them about the weather when s/he is in the kitchen. The user may also want that the lights of the bedroom are not automatically switched on and instead of waking up with the radio he may want to be woken up with relaxing music. To perform this type of evolutions, the tasks of a behaviour pattern can be changed.

Modifying the turn on the radio task

```
String newName="turn on relaxing music";
task_turnOn.setName(newName);
task_turnOn.setID("wakingUp_"+newName.replace(" ", ""));
task_turnOn.setServiceName("Music");
task_turnOn.setServiceMethodName("turnOn");
Argument musicType= taskModelFactory.createArgument();
task_turnOn.setTemporalRelationship(task_lighting.getTemporalRelationship());
```

Removing the lighting task

```
wakingUp.getRefinements().remove(task_lighting.getRefinedFrom());
```

Creating the new task

```
SystemTask newSystemTask= taskModelFactory.createSystemTask();
String nameNewTask="inform about the current weather";
newSystemTask.setName(nameNewTask);
newSystemTask.setID("wakingUp_" + nameNewTask.replace(" ", ""));
newSystemTask.setServiceName("Weather");
newSystemTask.setServiceMethodName("inform");
Refinement refinement= taskModelFactory.createRefinement();
refinement.setType(TaskRefinementType.TEMPORAL);
refinement.setRefinementFrom(wakingUp);
refinement.setRefinementTo(newSystemTask);
TemporalRelationship tempRel=taskModelFactory.createTemporalRelationship();
tempRel.setType(TemporalRelationshipType.ENABLING);
tempRel.setTemporalRelationshipFrom(task_makeCoffee);
tempRel.setTemporalRelationshipTo(newSystemTask);
```

Figure 12 Evolving which services are executed using OCean and MUTate

Figure 12 shows how to modify which services are executed in the *Waking Up* behaviour pattern using OCean and MUTate. As shown, the *turn on the radio* task has been modified to turn on relaxing music. The lighting task and its subtasks have been removed; therefore, it will not be executed anymore in the pattern. And a new task named *inform about the current weather* has been created in the pattern and has been related to the make coffee task using a temporal relationship of the *enabling* type.

Modifying context conditions. The conditions where the services must be executed may change over time. For instance, the user may want to be woken up a half an hour later (his timetable may change) and may want the heating to be turned on in the bathroom for 15 minutes instead of 10 (so that the bathroom is warmer when he takes a shower). To perform this type of evolution, the context conditions used in the specification of a behaviour pattern can be changed. Figure 13 shows how to change the context conditions of the *Waking Up* behaviour pattern using OCean and MUTate. As shown, the context situation of the pattern has been changed to be enabled at 8:15 and the temporal restriction of the relationship between the two first tasks of the pattern has been changed to 15 min.

```
                                                    Modifying the context situation
ContextSituation ContextSituation=wakingUp.getContextSituation();
ContextSituation.getRightPart().setValue("08:15");
```

```
                Modifying the relationship that relates the turn on bathroom heating task to the next task
task_turnOn.getTemporalRelationship().getTemporalRestriction().setMinutes(15);
```

Figure 13 Modifying context conditions using OCean and MUTate

Evolving the service execution plan. The user may want that the services of a behaviour pattern are carried out in another order. For instance, the user may want to have breakfast before taking a shower and that the coffee is made just before the radio is turned on, thus, the coffee would not be so hot when he takes it. To perform this type of evolution, the relationships between the tasks of a behaviour pattern can be changed. Thus, the first task to be automated should be the make coffee task, then the user must be woken up and after this, the bathroom heating must be switched on (the time for having breakfast is enough so that the bathroom is warm).

Figure 14 shows how to change the execution sequence of the services executed in the *Waking Up* behaviour pattern using OCean and MUTate. As shown, the first task is moved to be executed as the last task and the last task is moved to be executed as the first task. The relationships of these tasks have also been changed. In addition, the context situation of the pattern has also been changed because the pattern must start when the user must be woken up.

```
                                        Moving the bathroom heating task to the last task to be
                                                                automated in the pattern
task_bathroomheating.setTemporalRelationship(null);
task_lighting.getTemporalRelationship().setType(TemporalRelationshipType.ENABLING);
task_lighting.getTemporalRelationship().setTemporalRelationshipTo(task_bathroomheating);
```

```
                                        Moving the make coffee task to the first task to be
                                                                automated in the pattern
TemporalRelationship tempRel = taskModelFactory.createTemporalRelationship();
tempRel.setType(TemporalRelationshipType.ENABLING);
tempRel.setTemporalRelationshipFrom(task_makeCoffee);
tempRel.setTemporalRelationshipTo(task_turnOn);
```

```
                                                                Updating the context
                                                                      situation
ContextSituation ContextSituation=wakingUp.getContextSituation();
ContextSituation.getRightPart().setValue("08:00");
```

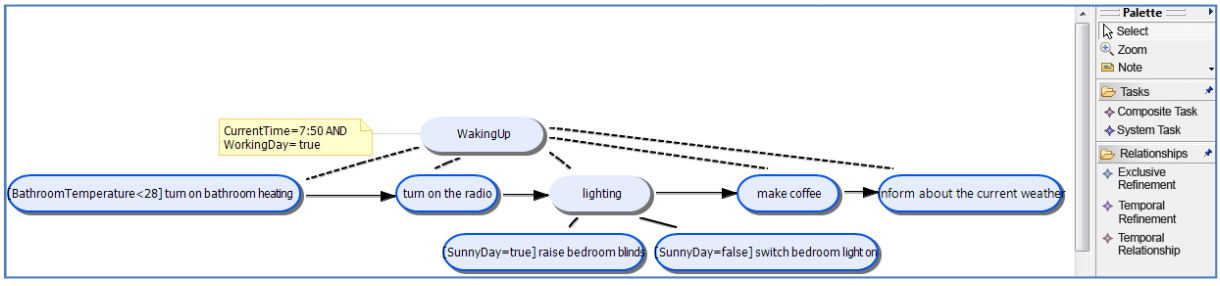
Figure 14 Evolving the service execution plan using OCean and MUTate

7. Tools for Evolving the Automated Behaviour Patterns

We present two tools that, using the mechanisms presented in the previous section, allow user behaviour patterns to be graphically evolved at a high level of abstraction and without stopping the system after deployment. The former is developed for being used by designers, while the latter is developed for being used by the end-users of the system. Both tools can be downloaded from http://www.pros.upv.es/art/. Finally, we analyse the different types of evolution that can be performed with these tools.

7.1 Tool for Designers

In this section, we present a tool that allows designers to graphically update the context-adaptive task model. This tool can be seen as a realization of the task model concrete syntax (presented in Section 4.2.) that corresponds to the abstract syntax defined in the task model metamodel (see Section 4.2.). The tool implements a model-view-controller architecture and has been developed using the EMF and GMF plugins of the Eclipse platform[8]. From the metamodel of the context adaptive task model specified in Ecore format (that is included in the EMF framework), we have generated java reusable classes for building an editor for the model. Using these classes and the facilities provided by GMF, the view and controller parts of the tool have been implemented.



[8] Eclipse Platform: www.eclipse.org

Figure 15. Snapshot of the graphical evolution tool

This tool is integrated with the mechanisms presented in the previous section. They constitute the model part of the model-view-controller architecture. Thus, the changes performed graphically by designers using this tool are automatically reflected in the models at runtime. They ensure that the changes are syntactically correct because these changes have to be in accordance with the metamodel definition. Note that designers can also use this tool to create the models at design time. Figure 15 shows a snapshot of the developed tool.

7.2 Tool for End-Users

In this section, we present a tool that allows end-users to evolve the automated behaviour patterns. Specifically, this tool allows users to perform the following actions in a user friendly way:

- 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, or 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. In addition, if users do not want certain patterns to be executed during a period of time, the tool also allows enabling or disabling them.

The concrete syntax used in this tool is based on wizards for end-users. These primitives used in the wizards are in accordance to the abstract syntax defined in the context model ontology and the task model metamodel. To be used by end-users, the wizards are built following the guidelines of Visual Programming approaches Visual Programming approaches [32], [33] and good-practices in End-user Development [34–37]. In order to develop these wizards, we have designed graphical interfaces, using the SWT (Standard Widget Toolkit) library. All the interfaces (see examples in Figures 16 and 17) share a similar structure and are displayed using a grid layout to facilitate user interpretation of the information shown. At the top of the interface, the different steps that should be performed in order to achieve the corresponding goal are shown. Thus, the user has an overview of all the steps and is guided step by step until the goal is completed. This design and the provided guidance improve the learning and memorization of the steps to be performed and keeps users from missing important ones [36].

The rest of the interface is divided into three frames:

1) The instruction area, which is shown at the top and provides users with instructions to help them to complete each step;

2) The working area, which is the area where users perform the corresponding step. The information that users need to be able to complete the step are popped-up s forms when needed. Thus, end-users just need to complete the information asked in the forms. These forms provide auto completion to reduce errors and user effort. Moreover, the working area also provides warning messages to warn users about errors committed when introducing the information (e.g. the user sets a text value rather than a numeric value) or to warn them about necessary information that has not been introduced.
3) The information area, which is shown at the bottom and provides users with feedback information in natural language about what the user is doing.

Following this design, we have developed interfaces for context specification and for pattern specification.

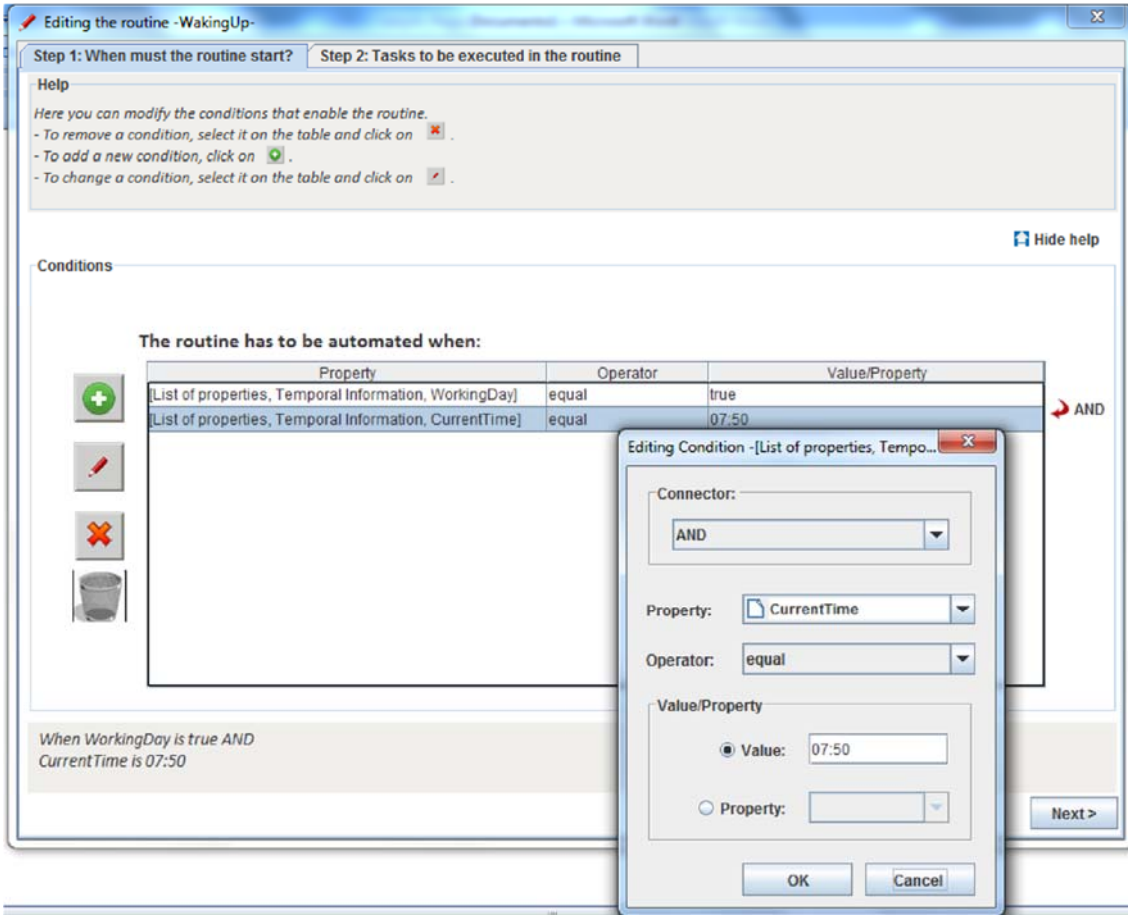


Figure 16 End-user interface for defining the context situation of the behaviour pattern

As representative examples, Figures 16 and 17 show the two steps that have to be accomplished to create a new behaviour pattern: context situation specification and task specification. Figure 16 shows the interface to specify the context situation whose fulfilment will trigger the execution of the pattern. Specifically, the figure shows a snapshot of the building of the context situation of the *WakingUp* behaviour pattern. The working area shows the specification of the

first condition: the *WorkingDay* property is equal to true. Once the user has specified the context situation of the pattern, the user must navigate to the task specification step which is shown in Figure 17. The working area allows users to specify the tasks that the pattern must execute. It shows the tasks that must be executed in the WakingUp pattern. The information area shows an explanation about the created tasks.

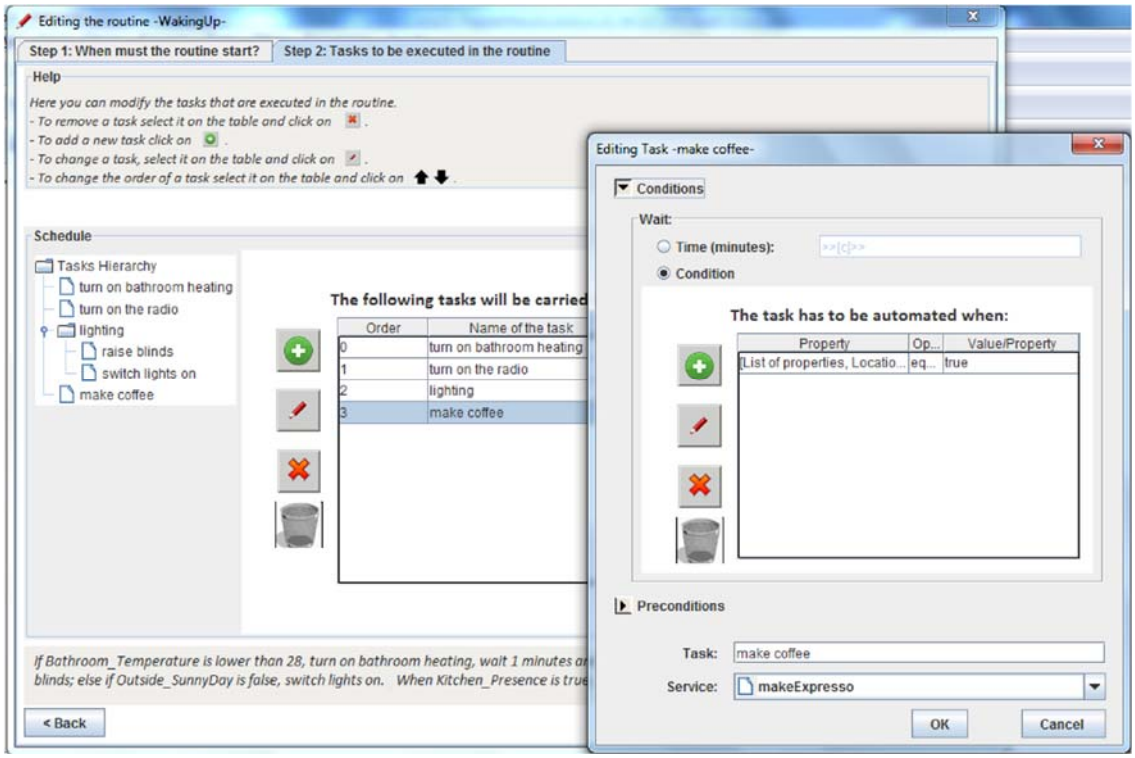


Figure 17 End-user interface for defining the tasks of the behaviour pattern

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. We have developed a prototypical tool that validates the consistency of the performed changes. Up to date, this tool validates the concordance of types, that all the system tasks have an associated service, that all the composite tasks have subtasks, etc. If any error is detected, the tool notifies the user about the possible mistakes so that they can be corrected. In the future, we plan to extend this tool to also check that there are no loops in the execution of the patterns and there are no inconsistencies with other patterns. To check if the execution of the services of a behaviour pattern can cause a loop (e.g., by making the context situation of the same pattern (or other patterns) to be satisfied again), the services provided by the AmI system should provide information about which operations perform contradictory tasks and also which context properties are modified by each service operation . Furthermore, we plan to provide the end-user tool with simulation capacities so that users can simulate the execution of the changed behaviour patterns to check beforehand if these patterns actually do what they want.

7.3 Supported Evolution

Using these tools, any change that respects the task metamodel and context ontology syntaxes can be performed to evolve the automated user behaviour patterns. Thus, new behaviour patterns can be created, and also those that are already specified can be modified and deleted. Using the same examples described in Section 6.2, we next show how the user behaviour pattern can be modified using these tools. Evolutions for creating new behaviour patterns or deleting them are performed in an analogous way.

Evolving the executed services. Figures 18 and 19 show how the executed services of the WakingUp behaviour pattern are changed: a new task named *inform about the current weather* has been created; the lighting task and its subtasks have been removed; and the *turn on the radio* task has been modified to turn on relaxing music.

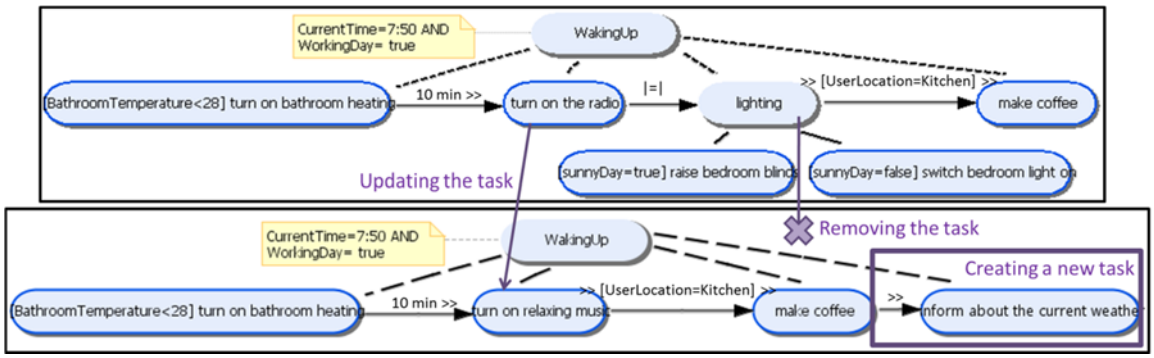


Figure 18 Evolving the executed services using the designer tool

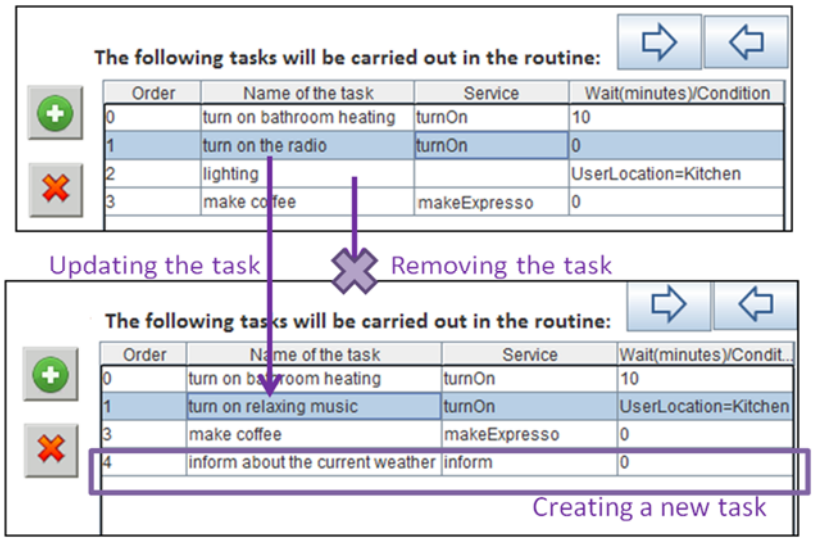


Figure 19 Evolving the executed services using the end-user tool

Figure 20 shows how the behaviour pattern is executed before and after performing these evolutions.

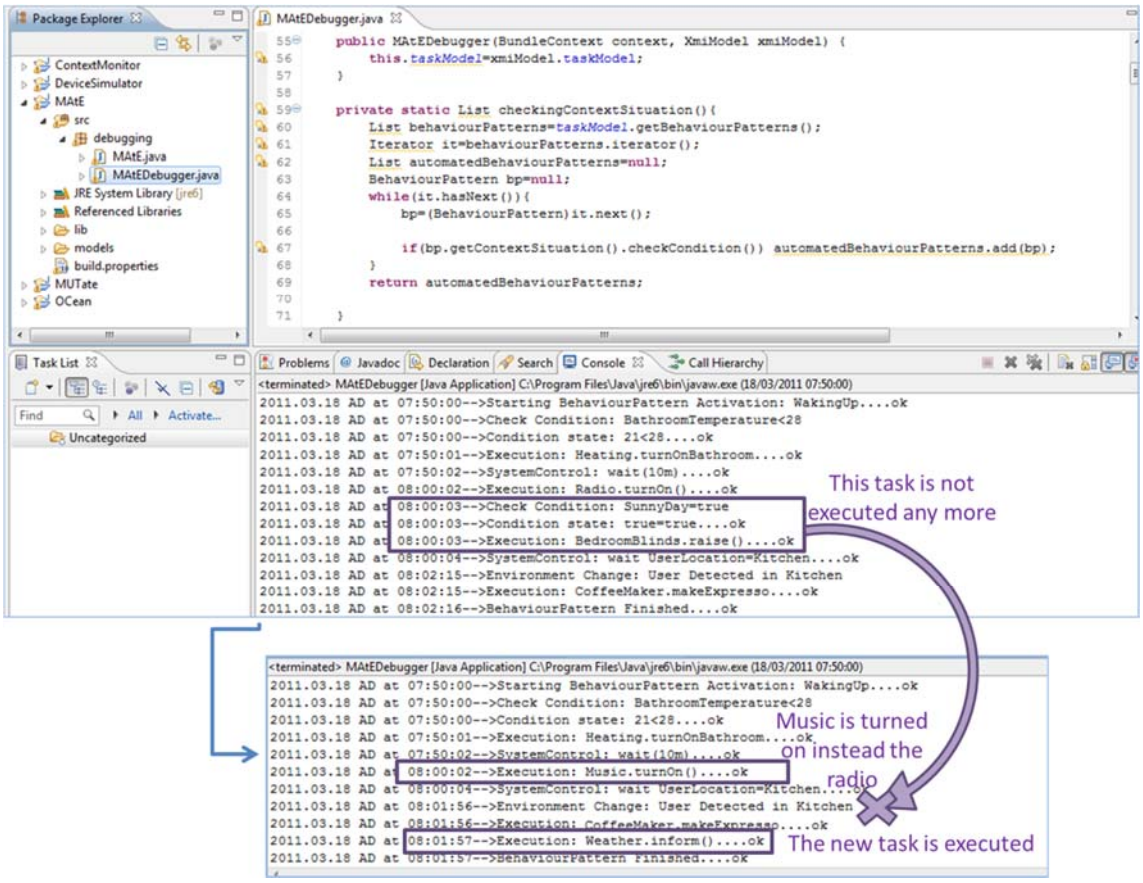


Figure 20 Execution traces before and after evolving the executed services

Modifying context conditions. Figure 21 and 22 show how the context situation of the specified behaviour pattern is changed to be enabled at 8:15 and how the temporal restriction of the relationship between the two first tasks of the pattern has been changed to 15 min.

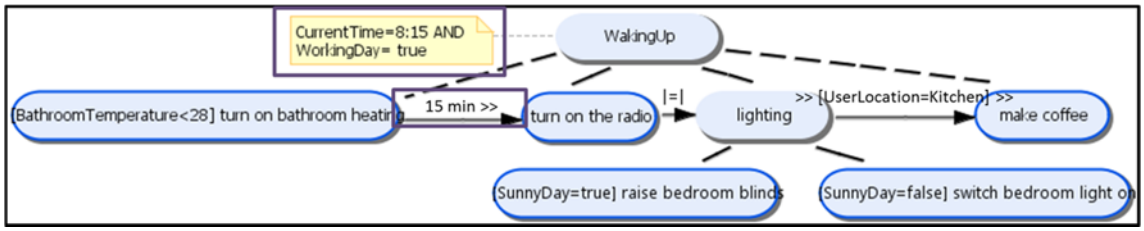


Figure 21 Modifying the context conditions using the designer tool

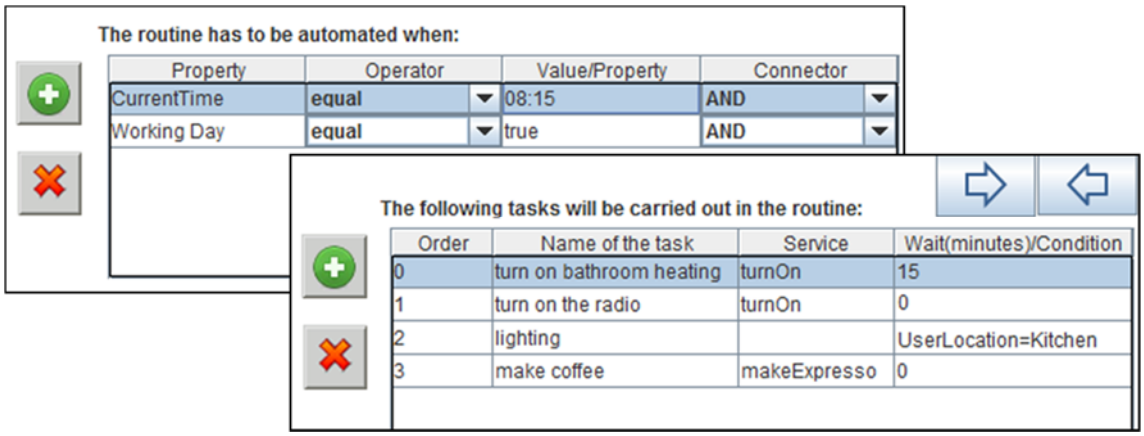


Figure 22 Modifying the context conditions using the end-user tool

Figure 23 shows how the behaviour pattern is executed after performing these evolutions.

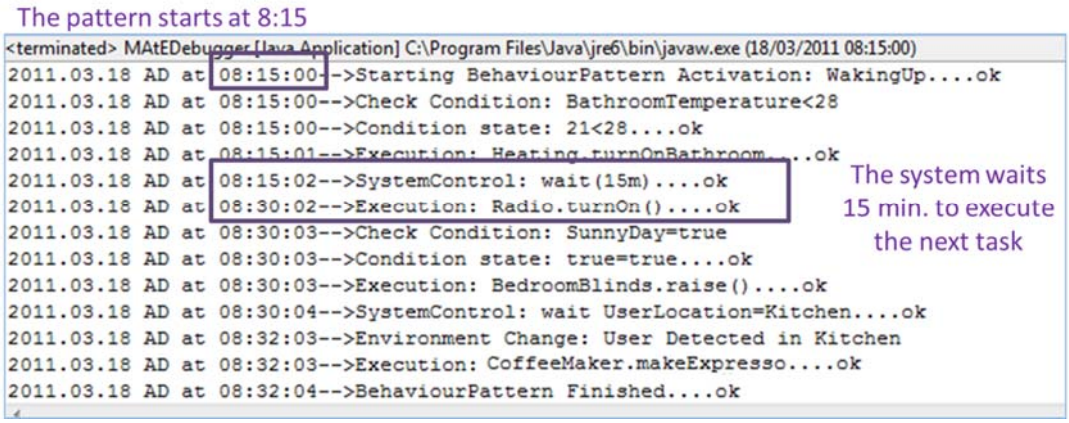


Figure 23 Execution trace after modifying the context conditions

Evolving the service execution plan. Figure 24 and 25 show how the service execution plan of the specified behaviour pattern is changed: the last task is now executed as the first task, and the first task as the last task; the context situation of the pattern has been changed to start at 8:00. Figure 26 shows how the behaviour pattern is executed after performing these evolutions.

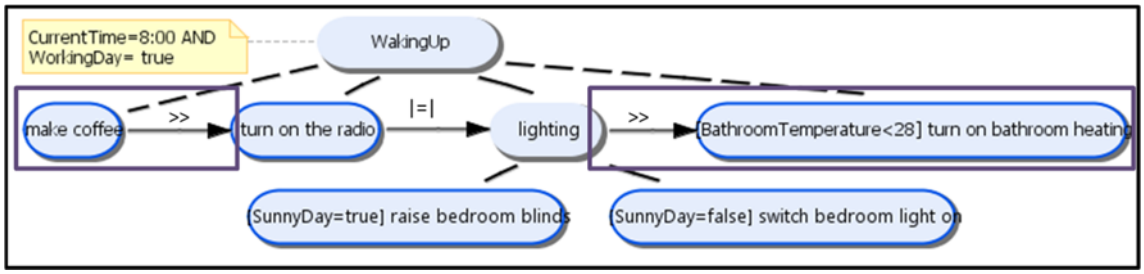


Figure 24 Evolving the service execution plan using the designer tool

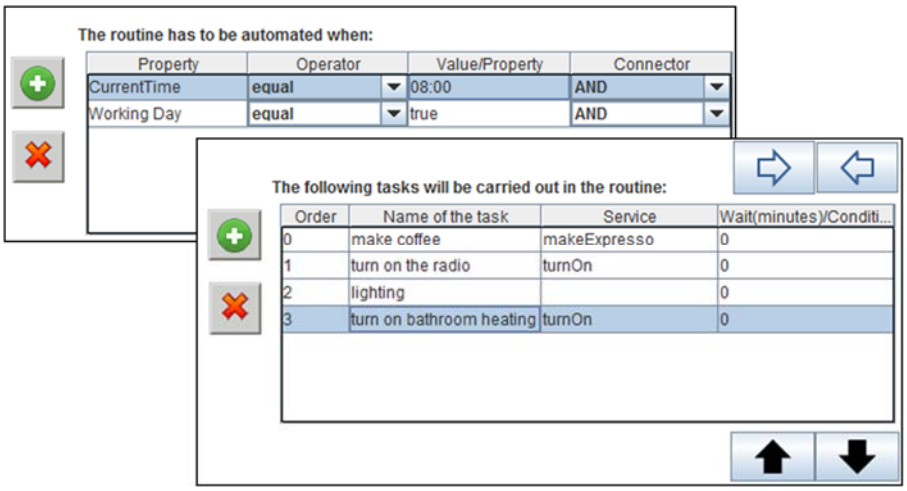


Figure 25 Evolving the service execution plan using the end-user tool

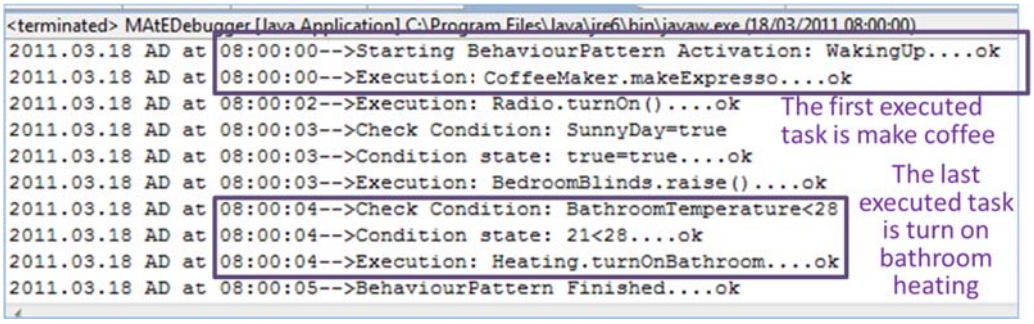


Figure 26 Execution trace after evolving the service execution plan

8. Evaluation of the Proposal

We have performed several evaluations in order to consolidate our proposal. First, we have tested the completeness and correctness of the automation and evolution of user behaviour patterns. Next, we have evaluated the performance of the system when using our approach in terms of time response and memory usage. Afterwards, we have performed and experiment to compare our MDD approach with an ad-hoc solution. Finally, we have evaluated the impressions of end-users when they used the end-user tool within several case studies.

8.1 Completeness and Correctness

To do these validations, we used computers with the following features: Intel Core 2 Duo P8400, 2.26 GHz processor and 4 GB RAM with Windows 7 Enterprise and Java 1.5 installed. In addition, we used an Equinox distribution as the implementation of OSGi.

In order to evaluate the completeness of the architecture we need to check that MAtE executes all the services required by a behaviour pattern before and after evolving it. In the same way, we must evaluate that all the behaviour patterns that must be triggered when a context situation is fulfilled are triggered before and after their evolution. In order to evaluate the correctness of the

approach we need to check that the services required by a behaviour pattern are all executed in the correct order and in the correct conditions. In the same way, we need to check that behaviour patterns are triggered if and only if the proper context situation is fulfilled.

To perform this, we based on the fact that the Context Monitor registers in the ontology each execution of the service associated to a system task (see Section 4 and 5.2), storing a history of the service executions. Thus, the proposed validation consists 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 (completeness) and in the correct order (correctness) before and after an evolution. It is clear then, that we must previously validate other aspects: that MUTate and OCean properly retrieve and save data, and that the Context Monitor registers service execution in a proper way.

In order to perform all these evaluations we used JUnit[9] tests. We developed a set of JUnit that allows us to evaluate: the proper behaviour of Ocean and MUTate; the correct behaviour of the context monitor; and the completion and correctness of MAtE. For instance, Figure 27 shows the JUnit method that evaluates the completeness of a behaviour pattern execution as a representative example. To perform this evaluation, we obtained 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 MUTate. Next, we executed the behaviour pattern through the method `executeBehaviourPattern(bp).` Afterwards, we retrieved the last registered automated operations from the ontology (i.e. we retrieved the individuals of the `AutomaticOperation` class) by interacting with OCean. We retrieved as many automated operations as the tasks the plan contained. Finally, we created an equal assertion to check if the automated operations retrieved from the ontology were the same as the tasks that the plan contained.

```
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);}
```

[9] www.junit.org

Figure 27. JUnit test example

These JUnit tests were applied to different user behaviour patterns and different scenarios. First, we evaluated that the behaviour patterns, triggered independently, were correctly automated; next, we evolved them; for each evolution, we applied again the JUnit tests and checked the correct execution of the evolved patterns. Then, we evaluated several scenarios combining the execution and evolution of several patterns (an example of these scenarios is shown in http://www.pros.upv.es/art). By analysing the results of the tests that failed, we identified and corrected some mistakes. For instance, we realized that the behaviour patterns dependent on time were executed repeatedly during one minute. This was because the system updated time every second and the smallest time unit considered in the behaviour patterns was minutes. Thus, the context situation of these patterns was continuously fulfilled until a minute went off. To solve this problem, we needed to use the same time unit in both cases. Considering that updating each second the context model could overload the system, we updated the context monitor so that time was updated every minute.

8.2 System Performance

Models are manipulated at runtime by MAtE. This manipulation is subject to the same efficiency requirements as the rest of the system. In particular, we evaluated how model operations impact overall system performance in terms of system response and memory usage.

System response. Model operations have to be efficient enough so that the system response is not drastically affected. In order to validate whether our approach scales to large systems, we quantified the temporal cost of the operation performed with randomly generated large models.

To test these operations we used the context model presented in Section 4.1 and an empty task model to be randomly populated by means of an iterative process. The context model was populated with 100 new context individuals in 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.

For each iteration, we tested all the model operations of MAtE 20 times and calculated the average temporal cost of each one. As an example, the operation of OCean with the highest temporal cost was the operation to get a context individual, which took 7 milliseconds with 100 individuals and 10 milliseconds with 6000 individuals. This operation has an initial cost caused by the access to the ontology (which explains the small time difference between 100 and 6000 individuals) and executes a SPARQL query whose time depends on the number of individuals. Figure 28 shows the temporal cost of the 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 temporal cost of the operations for getting, updating and deleting a task. These costs are very similar since all of them run 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.

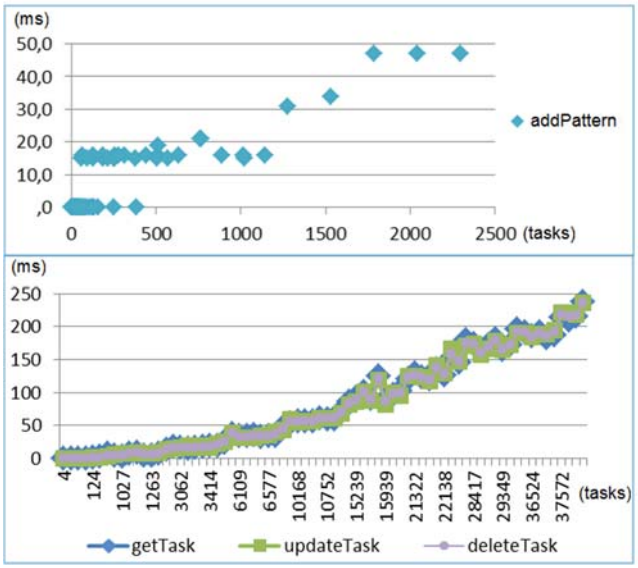


Figure 28 Temporal cost of accessing the task model at runtime

Memory usage. In order to interpret user behaviour patterns, MAtE needs to load them into memory. Thus, we measured the usage of memory in order to demonstrate that it is not excessive. To do so, we followed a strategy similar to the one used in the previous evaluation: we randomly generated behaviour patterns with a task structure that formed a perfect binary tree, varying its depth and the width of the first level. Next, we made MAtE to load them, and using the YourKit Java Profiler[10] tool, created memory snapshots to measure the size of the created behaviour patterns.

We measured the memory needed by behaviour patterns with a width from 1 to 10 and a depth from 2 to 6. We selected these values after evaluating different case studies developed during our research. Note that the larger pattern (width 10 and depth 6) includes 311 tasks, 159 temporal relationships and 320 refinements. This pattern is extremely large and very unlikely to be created in a smart home system. In particular, in the case studies that we have developed in this domain, we have created patterns with a maximum width of 6 and a maximum depth of 4.

Figure 29 shows the memory usage of the patterns with different width and depth. As an example, the larger pattern (width 10 and depth 6) measures about 110 Kbs. In a system with 50 patterns of this size, less than 5Mb of memory is needed to manage the task model at runtime.

[10] http://www.yourkit.com/

We found this result acceptable considering that all smart home systems analyzed and developed as case studies needed models that contain, on average, 15 patterns with a width of 5 and a depth of 3. In addition, note that smart home systems are currently deployed in powerful gateways that have from 256Mb to 1Gb of RAM memory, making the memory needed to manage task models to be a non-critical aspect for this domain.

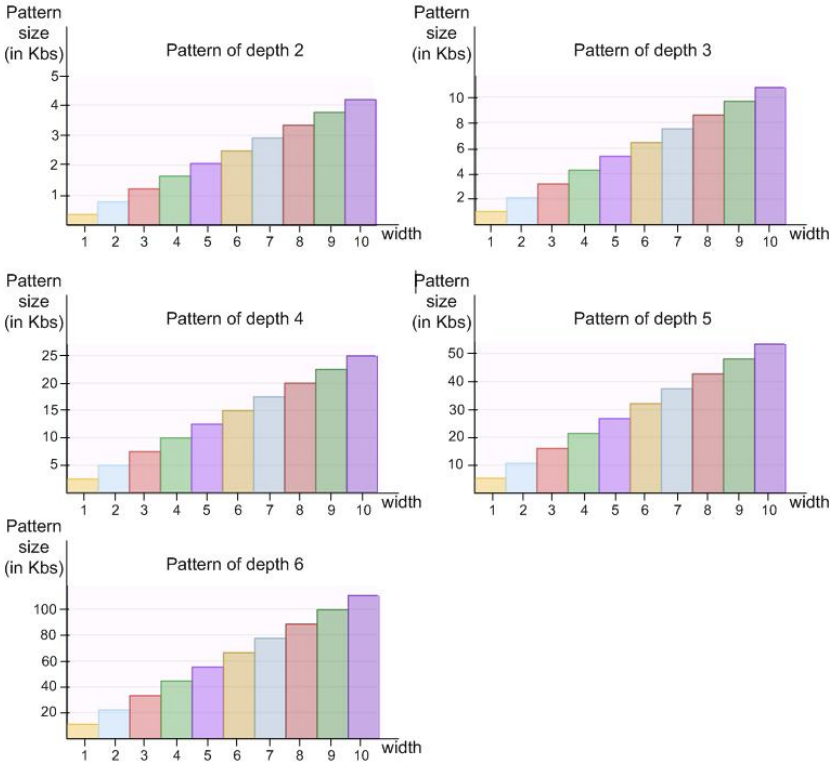


Figure 29 Memory usage for managing behaviour patterns at runtime

Finally, it is worth nothing that performance evaluation has been focused in the area of smart home systems. The performance of our approach in other domains may be different and may require some improvements as it is further stated in Section 10.2.

8.3 Usefulness of our MDD Approach

This section introduces the experiment that we have performed to show the usefulness of our MDD method in the development and evolution of automated user behaviour patterns. The aim of the experiment was to compare the usefulness measure obtained by our MDD proposed approach over the traditional development (i.e., hand-coding development). In this experiment we asked participants to develop and evolve user behaviour patterns by using both our MDD approach and ad-hoc development in Java/OSGi technology. Thus, the evaluation was focused only on computer engineers, since most end-users have not the knowledge and the skills required to use the Java/OSGi technology.

To do this experiment, we followed the guidelines presented by Kitchenham et al. [38] and Wohlin et al. [39]. Next, we present each experimental element.**Objectives.** According to the Goal/Question/Metric template [40] the objective of the experiment was to:

Analyse: Our MDD approach
For the purpose of: Evaluating its usefulness
With respect to: The traditional development (hand-coding)
From the viewpoint: Of designers
In the context of: Computer science developers

From this objective, the following hypotheses were derived:

- Null hypothesis 1, H10. The usefulness of our MDD approach for developing and evolving automated user behaviour patterns is the same as the traditional development.
- Alternative hypothesis 1, H11. The usefulness of our MDD approach for automated user behaviour patterns is greater than the traditional development.

Identification of variables. We identified two types of variables:

- Dependent variables: Variables that correspond to the outcomes of the experiment. In this work, usefulness was the target of the study, which was measured in terms of the following software quality factors: learning time, development time; deployment time; correcting errors time; and maintenance and evolution time [41].
- Independent variables: Variables that affect the dependent variables. The development method was identified as a factor that affects the dependent variable. This variable had two alternatives: (1) our MDD approach and (2) the traditional development.

Experimental context. The context of the experiment is the following:

- Experimental subjects. Ten subjects participated in the experiment, all of them being researchers in software engineering. Their ages ranged between 25 and 40 years old. The subjects had an extensive background in Java programming and modelling tools; however, they did not have experience in task modelling. Some subjects had knowledge about OSGi technology.
- Objects of study. The experiment was conducted using a case study similar to the running example used throughout the paper, i.e., the WakingUp user behaviour pattern (see Fig. 6). In order to shorten the evaluation process for both development approaches and to achieve similar implementations from user to user, we provided the subjects with a behaviour pattern example to guide the development of the WakingUp routine. Specifically, we

provided them with a traditional implementation of a behaviour pattern to support the shopping as well as its modelling using our MDD approach. The principal aim of this routine is to improve the everyday life of the users either by 1) addressing the shopping automatically if home delivery service is available, or 2) by providing users with information about the items that are needed and about the supermarket where the items can be bought otherwise. In addition, we provided the subjects with the drivers for the devices and services that they needed for developing the routines.

- Instrumentation. The instruments that were used to carry out the experiment were:
  - A demographic questionnaire: a set of questions to know the level of the users' experience in Java/OSGi programming, modelling tools, and task modelling.
  - Work description: the description of the work/activities that the subjects should carry out in the experiment by using our MDD approach and the traditional one. These activities were the following: (1) the development of the WakingUp routine; and (2) the modification of the Shopping routine (evolving the executed services by adding a new one, modifying the service execution plan, and changing the context conditions).
  - A form: a form was defined to capture the start and completion times of each activity. Some space was left after the completion time of each activity for additional comments of the subjects about the performed activity. If some activities were performed with interruptions, subjects wrote down the times every time they started and stopped carrying out the activity; thus, the total time was derived using these start and completion times.

Validity evaluation. The various threats that could affect the results of this experiment and the measures that we took were the following:

- Conclusion validity: This validity is concerned with the relationship between the treatment and the outcome. Our experiment was threatened by the random heterogeneity of subjects. This threat appears when some users within a user group have more experience than others. This threat was minimized with a demographic questionnaire that allowed us to evaluate the knowledge and experience of each participant beforehand. This questionnaire revealed that most users had experience in Java programming and modelling techniques. This threat was also minimized by providing the subjects with the Shopping behaviour pattern, which helped and guided them in the development of the WakingUp behaviour pattern. Also, our experiment may be threatened by the reliability of our measures; however, we used time measures, which are objective measures that are more reliable than subjective measures. In addition, the precision of the measures may have been affected because the activity completion time was measured manually by users using the computer clock. To reduce this

threat, we observed subjects while they were performing the different activities in order to guarantee their exclusive dedication in the activities and supervise the times that they wrote down.

- Internal validity: This type of validity concern is related to the influences that can affect the factors with respect to causality, without the researcher's knowledge. Our evaluation had the maturation threat: the effect that users react differently as time passes (because of boredom or tiredness). We solved this threat by dividing the experiment into different activities.
- Construct validity: Threats to construct validity refer to the extent to which the experiment setting actually reflects the construct under study. Our experiment was threatened by the hypothesis guessing threat: when people might try to figure out what the purpose and intended result of the experiment are and they are likely to base their behaviour on their guesses. We minimized this threat by hiding the goal of the experiment.
- External validity: This type of validity concern is related to conditions that limit our ability to generalize the results of the experiment to industrial practice. Our experiment might suffer from interaction of selection and treatment: the subject population might not be representative of the population we want to generalize. To deal with this threat, we used a confidence interval where conclusions were 95% representative. This means that if conclusions followed a normal distribution, results would be true for 95% of the times the evaluation is repeated.

Experimental design and procedure. We follow a within-subjects design where all subjects were exposed to every treatment/approach (MDD and traditional development). The main advantage of this design was that it allowed statistical inference to be made with fewer subjects, making the evaluation much more streamlined and less resource heavy [39]. In order to minimize the effect of the order in which the subjects applied the approaches, the order was assigned randomly to each subject. However, in order to have a balanced design, the same number of subjects was assigned to start with each approach. In this way, we minimized the threat of learning from previous experience.

The study was initiated with a short presentation in which general information and instructions were given. Next, the experiment started with a demographic questionnaire to capture the user's backgrounds. Afterwards, the work description and the form were given to the subjects and they started to develop the WakingUp behaviour pattern following the two kinds of development (MDD and traditional) in the indicated order for each user. For each activity of the development, the users filled in a form to capture the development times. Once the users developed the WakingUp behaviour pattern, they started to modify the Shopping behaviour pattern to evaluate the maintenance and evolution. For these activities, they also filled in the

form to capture the maintenance time. Specifically, the activities carried out in each part were the following:

- Traditional development: Prior to implementing the case study, we provided the subjects with the necessary tutorials and tools to learn the basics of the OSGi technology needed to develop the case study. The subjects also participate in a session where they implemented some guided examples in order to gain experience with the technology. Then, from the case study description, they started the implementation of the WakingUp routine taking the Shopping routine as example. Generally, they implemented the classes to support the routine functionality and the context management (sense the context from the devices and check context conditions). Then, the subjects deployed the system in the OSGi server. Once they achieved the execution of the code, they spent some time detecting and correcting code errors. Finally, we provided a set of requirement changes for the Shopping routine in order to evaluate the maintenance and evolution. In this activity, the subjects changed the provided implementation to support the new requirements. Then, the subjects deployed the changed routine and corrected the errors.
- Our MDD approach. Prior to implementing the case study, we provided the subjects with a tutorial where the modelling language and the designer tool were explained. The subjects also worked with some examples in order to gain experience with the model and the tool. Following the model-driven development presented in this paper, the subjects first designed the WakingUp behavior pattern according to the case study description. Then, they linked each task with the appropriate pervasive service. Then, the subjects saved the created models in the specific folder of the OSGi server where the software infrastructure was deployed. Once they achieved the execution of the models, they detected and corrected some modelling errors. Finally, the subjects applied the required changes into the design of the Shopping routine. To perform the changes, they opened the provided models with the designer tool and performed the changes according to the new requirements.

Analysis and interpretation of results. In this subsection, we analyse and compare the usefulness of both approaches based on the time used for learning, development, deployment, correcting errors, and maintenance. The results have been studied based on a time mean comparison and the standard deviation. Table 1 presents the descriptive statistics for each of the studied quality factors and Figure 30 shows the means comparison of the obtained measures for each approach.

Quality factor	Dev. method	Mean (hours)	Number of Subjects	Std. deviation (hours)
Learning time	Traditional	20.75	10	2.45

Learning time	MDD	11.24	10	1.64
Development time	Traditional	4.21	10	0.52
Development time	MDD	1.18	10	0.44
Deployment time	Traditional	0.92	10	0.65
Deployment time	MDD	0.24	10	0.17
Correcting errors time	Traditional	4.29	10	1.13
Correcting errors time	MDD	0.29	10	0.14
Maintenance time	Traditional	3.05	10	0.69
Maintenance time	MDD	0.24	10	0.05

Table 1. Descriptive statistics for each quality factor.

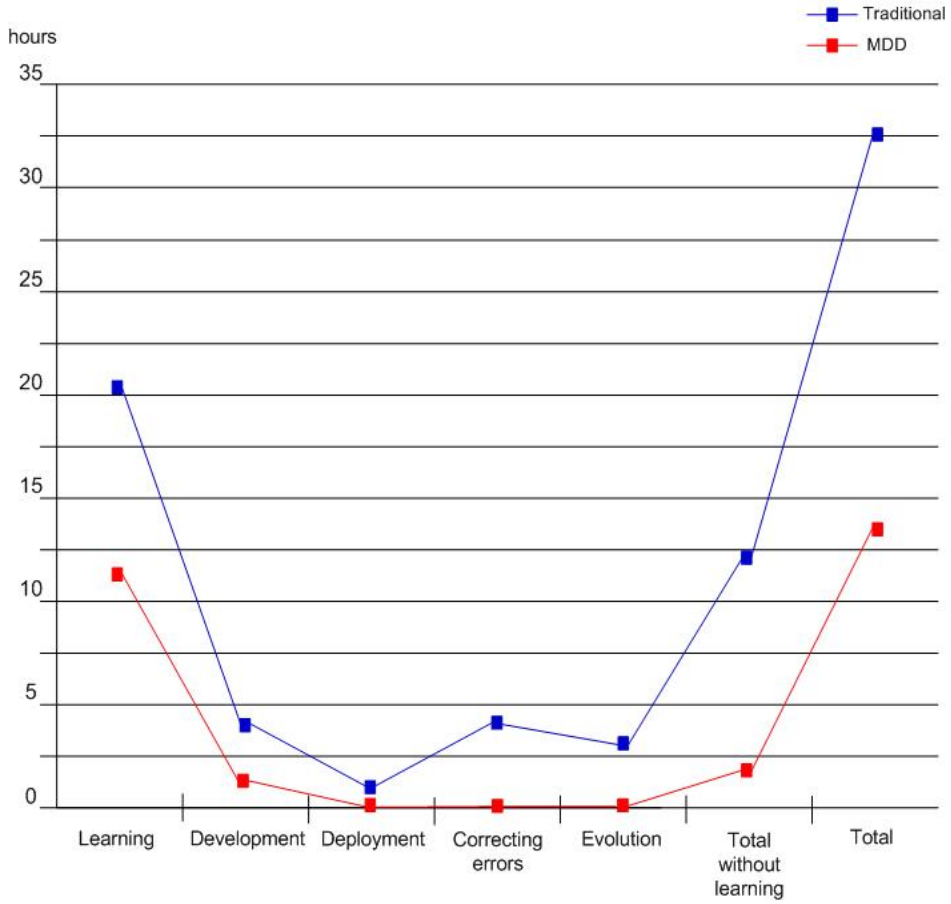


Fig. 30. Time mean comparison for development method usefulness.

Next, we provide a further analysis of the results for each measured software quality factor:

- Learning time. The learning activity in the traditional development took users between 18 and 25 h (distributed in several days). The subjects commented that having the implementation of the Shopping behaviour pattern helped them in this learning activity. Nonetheless, they needed to continue learning throughout the development of the case study. This activity following our MDD approach took the subjects from 10 to 14 h (also distributed in several days). Having the Shopping behaviour pattern modelling as an

example also helped them to understand the modelling language. Thus, the learning time was significantly higher in the traditional development. This is because technologies like OSGi are complicated and learning them takes longer than learning our modelling language. The MDD approach provides technology-independent, high-level primitives to specify the system. Also, more dispersion was found in the learning time following the traditional development (std. deviation = 2.45), because the subjects that had knowledge about OSGi took much less time than the subjects without this background. It is worth noting that this activity was performed with interruptions since it took a significant amount of time for some subjects.

- Development time. The development time following the traditional development differed according to the users implementation experience, ranging from 3.25 (the most experienced subject) to 5 h. Following the MDD approach, the development activity ranged from 75 min to 2.20 h. The difference between the two approaches was high since developing the routine traditionally was more complex and difficult for the subjects (because they had to hard-code all the user behaviour logic manually). The MDD approach allowed subjects to focus on satisfying user behaviour requirements instead of solving technological problems. Note that by following this approach, none of the subjects had to implement anything to manage the context information since it is automatically managed by our software infrastructure and the context ontology. Regarding the standard deviation, it was low for both development approaches (see Table 1) indicating that development times tended to be close for each development approach.
- Deployment time. In this activity, the subjects deployed the system in the OSGi server. In the traditional development, the subjects that had already worked in OSGi deployed the routines in a few minutes (from 5 to 10 min). However, for the rest of subjects, the deployment took more time (from 40 to 70 min). Although they had work with this issue in the preliminary workshop, they still had little expertise to package the code into bundles and start them in the OSGi server in the appropriate order. The standard deviation was (0.65), The deployment of the system using the MDD approach consisted on saving the models and pressing the run button. For this reason, the standard deviation was very low (0.17).
- Correct errors time**.** In the traditional development, the subjects spent from 3 to 6 h to detect and correct errors in the code. The common errors were having infinite loops or wrong conditions in the temporal operators of a task. For example, one subject specified that the WakingUp pattern should start when WorkingDay was false. The right condition was WorkingDay=true but it took him quite long time to detect this error. Conversely, with the MDD approach, the subjects spent from 15 to 30 min to detect and correct some modelling errors. Five subjects had problems in the creation of the task hierarchy because they created

more tasks than needed. Eight subjects had errors in the context conditions associated to the tasks and the temporal relationships. This was because the names of the context properties have to be the same as the names specified in the context ontology. However, most of the subjects commented that the correction of faults and the performance of changes in the user behaviour patterns could be carried out more easily using the graphical models than analysing the code, since they could intuitively locate what to change. The obtained standard deviation shows more dispersion following the traditional development (see Table 1), indicating that some subjects had more difficulties than others for correcting the errors.

- Evolving time. With regard to the traditional development, most of the users commented that it was not easy to modify the code that they had not implemented. This activity took them from 2 to 4h. Changing the models following the MDD method was easier and quicker for the subjects since they already knew the semantics of the models. This activity took less than 15 min for all the subjects (very low standard deviation obtained). This is because updating a user behaviour pattern (such as adding, modifying or deleting tasks or context information) is as easy as modifying the specified pattern in the graphical tool to make it fit the new requirements.

With our MDD approach, the subjects took 1.95 h to develop the case study (without considering the learning time), whereas with the traditional development the subjects took 12.47 h. Therefore, the process for automating and evolving user behaviour patterns is more efficient using our MDD approach than using the traditional one. In order to verify whether we can accept the null hypothesis, we performed a statistical study called paired T-test using the IBM SPSS Statistics V20[11] at a confidence level of 95% ($\alpha = 0.05$). This test is a statistical procedure that is used to make a paired comparison of two sample means, i.e., to see if the means of these two samples differ from one another. For our study, this test examines the difference in mean times for every subject with the different approaches to test whether the means of the traditional development and our MDD approach are equal. When the critical level (the significance) is higher than 0.05, we can accept the null hypothesis because the means are not statistically significantly different. For our experiment, the significance of the paired T-test for the total time means is 0.000 (calculated using the IBM SPSS Statistics), which means that we can reject the null hypothesis H10 (the usefulness when using our design method for designing behaviour patterns is the same as when using the traditional design). Based on this test, we have given strong evidence that the kind of development influences the usefulness. Specifically, the usefulness using our MDD approach is significantly better than using the traditional one, i.e., the mean values for all the measures are lower when using the MDD approach; thus, the

[11] Statistical analyses using spss, http://www.ats.ucla.edu/stat/spss/whatstat/whatstat.htm#1sampt

alternative hypothesis H11 is fulfilled: the usefulness when using our approach for designing behaviour patterns is greater than when using the traditional design.

8.4 End-User tool evaluation

This evaluation consists in measuring the satisfaction of using the end-user tool presented in Section 7. We proposed to 18 users (10 men and 8 women, which ranged from 26 to 57 years of age) to participate in the evaluation of the end-user tool presented in Section 7.2. All these end-users had actively participated as clients in the model specification of the case studies used in the previous evaluation. These participants covered a wide variety of professions including engineers, an administrative staff, teachers, housewives, nurses, a farmer, and students. 10 of them had a medium-high level of computer knowledge and 8 had basic computer knowledge.

In order to evaluate the end-user tool, we used the Post-Study System Usability Questionnaire (PSSUQ) published by IBM in [42]. This questionnaire is a 19-item instrument for assessing user satisfaction with system usability. In addition, the items of the questionnaire ask users: if the tool was easy to learn to use, which allows us to measure the learnability of the tool, i.e., how easy is to learn to use the tool; and if they were able to efficiently complete the tasks and scenarios using this tool, which allows us to evaluate the applicability of the tool (i.e., how easy is to evaluate the models once one knows how to use the tool).

The participants evaluated the prototype developing a series of scenarios previously designed using the end-user tool. These scenarios included the following tasks:

- enable/disable a behaviour pattern;
- delete a behaviour pattern;
- add a new preference and change its value;
- modify the context situation of a behaviour pattern;
- modify the tasks of a behaviour pattern (service in charge of executing them, waiting time between tasks, etc.);
- add a new simple behaviour pattern (without composite tasks): create the context situation and specify the tasks to be executed;
- add a new complex behaviour pattern (with composite tasks): create the context situation and specify the tasks from more complex to simpler.

For instance, users were asked to carry out the scenarios described in Section 6.2 and 7.3. We arranged several sessions in which the users first explored the tool to get a feel for its functionality and then carried out these scenarios under our supervision. Once the users completed the scenarios, they filled out the PSSUQ. According to the questionnaire, whose

scores were on a scale of 1 (the highest score) to 7 (the lowest score), the main results obtained from this evaluation were the following:

- With regard to the learnability of the tool, the participants of the experiment commented that the tool was easy to learn to use. Specifically, 55.55% of the participants gave the tool a 1 (the highest score) and the rest a 2. They said that the structure of the interfaces organization and the guided steps helped them to easily learn how to perform the activities.
- Concerning the applicability, 66.67% of people involved in the evaluation perceived the evolution of the automated patterns as very easy, giving this item a 1. However, from the participants with basic computer knowledge, 22.22% gave this item a 3 and the rest a 4. These participants commented that modifying a pattern was not very difficult because they had participated in the modelling of the behaviour patterns and they knew them. However, they found difficult the activity of creating a new complex behaviour pattern because it took them a lot of effort to organise the task hierarchy using context conditions. Also, during the development of the scenarios, we also observed that designing the context conditions was more difficult for some users than what we had previously expected. In spite of this, the interface design helped them to correctly specify the conditions.
- Regarding tool usability, the PSSUQ revealed that the tool was clear enough. Most users found the interface to be friendly and easy to use. The worst score obtained from the questionnaire was for the questions that determine the information quality of the message errors. Users commented that the help messages that were shown in the interfaces helped them to complete the tasks; however, when they committed an error, the error messages were not clear enough to correct them. To improve this aspect, we are currently working to make these messages clearer for users by showing examples of solutions to correct the possible errors. Regarding what users did not like or they would change, some users, essentially those with basic computer knowledge, commented that designing the context conditions was still difficult for them. They commented that they would like to have a list of predefined conditions in which they could change some parameters (i.e., when it is -day of week-, at -time-, in a working day, when nobody is at home, etc.). After explaining them how a condition is formed in depth, we observed that the interface design helped them to correctly specify the conditions since it provided all the context properties and the operators that they could use and helped the users to fill the values. However, we plan to extend the tool to support predefined conditions. In addition, some users commented that they preferred forms instead of filling out tables,

because they said that forms would facilitate to change the behaviour patterns. We plan to support this aspect in further work.

- Regarding the overall satisfaction of the tool, it had a result for average of 2.158 according to the questionnaire, which is a quite acceptable mark.

9. Related Work

In this section, we revisit some of the most popular and relevant related works found in the literature and compare them with our approach. We present two kinds of related works: proposals to automate and evolve user behaviour patterns, and proposals that perform evolution by using models at runtime.

9.1. Behaviour Pattern Automation Research

In this work, we have presented an approach for automating and evolving user behaviour patterns by using context-adaptive task descriptions. Previous approaches for dealing with this challenge can be grouped into: machine-learning approaches, rule-based context-aware systems, and task-oriented computing systems.

Machine-learning approaches use algorithms to infer user behaviour patterns from historical data and to automate them. Some outstanding examples are: MavHome [43] and iDorm[44]. The MavHome project, extended by the CASAS project [45], uses prediction algorithms to identify common sequential patterns from data captured from the sensors of a smart home. From this learning, Mavhome and CASAS built 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 Markov model can be extended with new states or the system can be rebooted to obtain an improved Markov model using a new set of observations. The iDorm project predicts 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 as necessary. Thus, the evolution of these models is performed by rebooting the system or at a low level of abstraction. In contrast, in this work, the evolution can be performed at runtime and at the modelling level using concepts of a high level of abstraction, such as task or behaviour pattern, instead of adding states or rules as the approaches presented above. In addition, these approaches present some limitations:

1) They require a great amount of training data (the cold-start problem) [46–48];

2) Lack of knowledge about user performed tasks and user desires may lead to automating tasks for which the users may not want automation or reach generalizations in such a way that the automation becomes a burden on the user;

3) They can only reproduce the actions that users have executed in the past in the same way users executed them.

In our approach, we tackle these problems. We provide the system with a set of tasks that are automated since it starts to run, providing a solution for the cold-start problem (see [48] for more detailed information). In addition, tasks to be automated are analyzed previously by analysts with user participation. Therefore, only those tasks that users do want are automated. Moreover, both users' desires as well as time and energy concerns can be taken into account; and tasks can also be automated regardless of whether or not the users have performed them in the past.

Rule-based context-aware approaches program rules to automate user actions when a certain context condition arises. Some outstanding examples are [49] and [50]. In [49], the system is composed of independent agents, each of which comprises a set of rules that encode reactions to context states. Each rule follows the template *When <triggers>, if <conditions>, then <action>*. In [50], the authors propose situation and preference abstractions that are used by ECA rules to automatically trigger certain actions. Both works allow users to configure the specified rules by modifying user preferences; however, these approaches do not support the evolution of the rules or favour user participation in the design of the rules. In contrast, in our approach, the automated routine tasks are described and managed by using the task model. This model facilitates the participation of users in its design and evolution by using concepts close to them [51]. Also, rule-based techniques generally require large numbers of rules which have to be manually programmed [52] and are not appropriate for automating user complex tasks as the behaviour patterns described in Section 4.2.

Task-oriented computing systems use task modelling to facilitate the interaction of users with the system since the concept of task is more understandable for end-users [51]. These systems have proved that task modelling is effective in several fields such as user interface modelling [53–55], and assisting end-users in the execution of tasks through service provisioning and resource allocation [56–58]. These works show the growing usage of task modelling and its remarkable results and possibilities to model system behaviour. However, they do not attempt to automate user behaviour patterns. Therefore, the proposed task models do not provide enough expressiveness (such as specific relationships between tasks, context situations, etc.) or runtime support for their execution and evolution.

9.2. Software Evolution by Using Models at Runtime

In our approach, we confront system behaviour evolution by evolving models at runtime. In this section, we study some of the most important works that use models at runtime for software evolution.

Oreizy et al. [59] were pioneers in their adoption of an architecture-based model to support runtime software evolution. Their approach emphasizes the role of software connectors in supporting runtime change in the system configuration. Connectors are explicit architectural entities that bind components together and act as mediators between them. The runtime architectural model describes the interconnections between components and connectors as well as their mappings to implementation modules. Thus, to perform system reconfiguration (adding, removing, or reconnecting a component), they alter the connector bindings that mediate component communication, maintaining the correspondence between the model and the implementation.

Floch et al. [60] promote the use of architecture models to support the development of adaptive mobile applications. They use adaptation policies expressed as high-level goals to be achieved. At runtime, the configuration of the system is optimized with respect to these goals. This reconfiguration is determined by comparing the actual running system with new architectural variant models based on a utility function.

Morin et al. [61] propose a combination of 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.

Cetina et al. [62] propose the reuse of design variability models at runtime for supporting the self-configuration of systems when triggered by changes in the environment. A context monitor checks if certain context conditions are fulfilled. If so, the models are interpreted to generate the needed reconfiguration actions to modify the system architecture. Afterwards, the variability models are updated accordingly to reflect the changes in the system architecture.

Garlan and Schmerl [63] use architecture-based models during runtime for system monitoring, problem detection, and repair. Their approach monitors the runtime behaviour of an executing system. The extracted information is abstracted and related to architectural properties and elements in an architectural model. Thus, if there are changes in the architectural model, a rule evaluation is triggered to determine whether the system is operating within an envelope of acceptable ranges. Violations of rules are handled by a repair mechanism that adapts the architecture. Finally, architectural changes are propagated to the running system.

Blumendorf et al. [64] 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.

None of the above approaches deal with the evolution of behavioural models. In addition, to be able to evolve the system, most of these approaches need either to synchronize models and the code that implements these models, or to synchronize different models by performing model to model transformations. In contrary, we support the evolution of the automated behaviour patterns through the evolution of the task model and context model at runtime. Since these models are the only representation of the automated behaviour patterns, we do not need to synchronize any other software artefact.

Contrary to the above approaches, MOCAS [65] deals with building self-adaptive component-based systems. It specifies the structure of the components using a UML profile whereas their behaviour is specified using UML state machines that are also used to describe adaptation policies. A MOCAS component embeds at runtime the state machine describing its behaviour, which is executed by the MOCAS engine, a Java library. The components are installed in a state-based container to become adaptive. The container intercepts and mediates adaptation to its wrapped component. This approach is specific for component-based systems whose behaviour and adaptation can be described by UML state machines. In addition, it is more focused on self-adaptation, and therefore, does not provide evolution tools that allow designers or end-users to evolve the system behaviour.

In [66], authors present a Graph-based Runtime Adaptation Framework (GRAF) to facilitate software behaviour adaptation by using runtime models that reflect the current state of the adaptable software's source code. This approach also takes advantage of model interpretation for facilitating adaptation; however, the main purpose of this approach is to provide runtime adaptation to Java applications already implemented, which differs from the purpose of the presented work, in which the adaptive components (the behaviour patterns), are only represented in the models (not in source code). In contrary to our work, this approach makes necessary reification and reflection capabilities for modifying compiled code at runtime in order to synchronize it with models.

10. Discussion

The use of models at runtime [67] is an emerging paradigm within Model Driven Development that promotes the use of models as runtime software artefacts. Models are proposed to be used

at runtime to monitor and verify particular aspects of runtime behaviour, implement self-* capabilities (e.g., adaptation technologies used in self-healing, self-managing, self-optimizing systems), and support human-driven adaptation. In this work, we use models at runtime in order to adapt the execution of AmI services to user behaviour patterns.

To be able to use models at runtime, we must provide software infrastructures that interact with them at runtime. To achieve this, we have developed MAtE, which is in charge of interpreting models and executing services accordingly. Along the paper, we presented some tools that show how MAtE allows us to evolve AmI system's behaviour by evolving models at runtime. Next, we summarize the main benefits of this approach and the challenges that the use of the execution of models raises.

10.1. Advantages of the Presented Approach

The approach presented in this work provides the following main advantages:

Evolution at a high level of abstraction. As we can see in previous sections, the system behaviour that is in charge of automating the execution of AmI services can be evolved by using concepts such as user behaviour patterns, tasks, or context situations. For instance, in order to make sure that an automated behaviour pattern executes one service instead of another we just need to replace the corresponding task; in order to change the order in which services must be executed we just need to modify temporal relationships; in order to change the situations in which services must be executed we just need to change a context condition. By following ad-hoc solutions, this evolution must be done by re-encoding complex logic structures, which constitutes a time-consuming and error-prone activity.

Post-deployment evolution without stopping the system. The proposed models are the only runtime artefacts that represent the user behaviour patterns to be automated. This ensures that as soon as models are evolved the system behaviour is evolved. We have shown in this paper how this aspect facilitates the creation of tools that allow us to evolve the models (and then the system behaviour) at runtime, without stopping the system. In addition, models are decoupled from the pervasive services' implementation since the models just use identifiers for reference the services. This also facilitates the evolution of the implemented services since the models are independent of their internal implementation.

Controlled evolution. The software evolution proposed in this paper is performed at the modelling level. This means that any evolution that can be done must satisfy the restrictions defined in the metamodels, i.e. we can evolved the coordination of AmI services according to the actions that are allowed by the metamodels. For instance, we cannot create a task that does

not belong to a behaviour pattern. This provides a valuable mechanism to guarantee that evolution is done in a safe way according to specific restrictions. It is true that they are syntactical restrictions and they do not control semantic restrictions. As it is explained in the future work section, we are working on developing simulation tools in order to check that evolutions produce the results that are expected to obtain.

Support to develop evolution tools. We have presented MUTate and OCean, which provide high level Java constructors that facilitate the evolution of the context model and the context-adaptive task models at runtime. These mechanisms, together with the presented concurrency module, facilitate the development of tools to evolve user behaviour patterns.

End-user evolution. This benefit is closely related to the previous one. As a proof of concept, we have developed an application that makes use of MUTate and OCean in order to evolve the system behaviour. This tool is focused on allowing the end-users to perform the evolution by themselves.

10.2. Challenges

Throughout the development and validation of the proposed architecture, we detected some challenges that arise to correctly support system behaviour evolution at runtime.

Model checking and validation. The use of model-based techniques provides great facilities for evolving system behaviour at runtime. However, to achieve the evolution of the system ensuring consistent system behaviour, the changes that have to be applied should be validated before performing them.

To perform these validations in our approach, model checking techniques can be applied. For instance, the developed tools already apply model checking by validating constraints that the models should fulfil, such as the following one (a full description of the model constraints that are checked can be found in [22]): when a task is refined only one type of refinement can be used to obtain child tasks. In OCL:

```
Context CompositeTask
Inv: self.refinements -> forAll(t1, t2 |t1.type = t2.type)
```

However, it is important to note that the presented task models are adaptive models, which makes their behaviour more complex to validate since it changes at runtime in response to context changes. Therefore, more complex validations are needed. For instance, it would be necessary to check that the modelled patterns do not produce loops when they are executed or that a simultaneous execution of two patterns does not produce undesired behaviour. To verify

and validate these aspects, other model-checking approaches specifically proposed for such models, such as the presented in [68], can be applied.

Model-based Simulation. Although model-checking techniques are very powerful tools for validating the syntactic correctness of the changes applied to models, other validation techniques are needed to validate the semantic correctness, i.e., to check beforehand if the changed patterns actually do what the users want. To achieve this, model-based simulations can be performed: they can allow end-users to observe the behaviour without taking any risks.

As explained in Section 7.2., we plan to provide the end-user tool with simulation capacities so that users can simulate the execution of the changed behaviour patterns. An outstanding example that could be applied with that purpose in our work is the one presented in [26]. It uses workflows to control the device operations and the interactions among the devices of a household. These workflows are executed by a workflow engine, in a similar way as MAtE executes the context-adaptive task models. Each workflow, the devices involved in it, and their environmental effects are modelled using Petri Nets. These Petri Nets are updated with environmental parameters obtained via real sensors and/or interrogating devices for their observable states at run-time. Thus, the Petri Nets act as mirror models for reasoning and predicting the effects of the workflow execution. Users or designers can run the Petri Net simulation to predict environmental effects before (or while) a workflow is (being) executed.

Performance Improvements. In the evaluation of our approach we have validated that the performance of model manipulation is acceptable when compared to the performance of the devices and communication networks usually found in the Smart Home domain. However, we may need to improve this performance if the approach is applied in other domains that require faster system response or less usage of memory, such as industrial systems or larger systems.

With regard to system response, parallelism techniques can be applied to improve it, so that multiple instances of MAtE can be executed to interpret behaviour patterns in parallel. However, this should face the problem of accessing models by two or more engines in a concurrent way. The concurrency module presented in this work should be improved to consider this point.

With regard to memory usage, a lazy load strategy can be implemented if memory is a critical resource. With this strategy, user behaviour patterns should not be completely loaded into memory. Only the root tasks, which represent the behaviour pattern, and the context situations, which indicate when a behaviour pattern should be executed, are required to be initially loaded. When a behaviour pattern has to be executed the rest of its tasks, refinements and temporal relationships can be loaded, interpreted and executed. Once it has been executed its definition can be removed from memory.

11. Conclusions and Future Work

In this work, we have presented and evaluated a novel approach for confronting the challenge of evolving automations related to user behaviour patterns. This approach is based on model interpretation at runtime, which allows us to evolve system behaviour based on the premise that the a*utomated user behaviour patterns are evolved by evolving the models.*

This way of facing the problem of system evolution allows us to provide mechanisms to: (1) evolve automations by means of high level concepts at runtime and (2) allowing designers and end-users to perform the required evolution.

To achieve this, we have proposed a context model and a context-adaptive task model that allow behaviour patterns to be specified in an abstract way. These models are interpreted at runtime by MAtE, an engine capable of executing the behaviour patterns as specified in the models. To evolve these automations we provide:

(1) High level mechanisms that allow models to be evolved by using their own modelling language.

(2) A toolkit that allows designers to specify and evolve the automated behaviour patterns by graphically modifying the models.

(3) A toolkit that allows end-users to evolve their automated behaviour patterns by using user-friendly interfaces. Thus, if user behaviour patterns undergo changes after system deployment, end-users can evolve them by themselves without having to stop the system.

As future work, we plan to work on extending our approach to support the use of machine-learning algorithms. We believe that these algorithms can considerably automate the evolution process. When the system is running, the context monitor stores the user actions in the context model. Machine-learning algorithms can use this information to automatically infer new behaviour patterns. To achieve this, we are extending the context monitor to provide an adequate Java interface that allows these algorithms to properly obtain this information from the context model. For not being disruptive for users, we also plan to extend the end-user tool so that it periodically shows users the inferred behaviour patterns and allows them to simulate the patterns' execution. Thus, users can modify the patterns and add them to the system if they so desire.

Regarding the application domains, our approach can be applied in a wide variety of smart environments, because our software infrastructure is implemented using Java and OSGi. We are currently developing a case study for automating and improving the irrigation and fertilization in an orange field. Using our approach, these tasks could be performed automatically and in a

more efficient way. Some of the behaviour patterns that have been identified to be automated are the following: *periodicIrrigation* and *periodicFertilization* to periodically irrigate and fertilize the land according to its humidity and the season; and *frostSecurity,* to irrigate the land when a frost has been predicted in order to prevent the oranges from freezing.

References

[1] M. Weiser, "The Computer of the 21st Century," *Scientific American*, vol. 265, pp. 66–75, 1991.

[2] E. Serral, P. Valderas, and V. Pelechano, "Context-Adaptive Coordination of Pervasive Services by Interpreting Models during Runtime," *The Computer Journal*, vol. 56, no. 1, pp. 87–114, 2013.

[3] S. A. Ajila and S. Alam, "Using a Formal Language Constructs for Software Model Evolution," *Third IEEE International Conference on Semantic Computing (IEEE-ICSC 2009)*. Berkeley, CA, USA, pp. 390–395, 2009.

[4] K. Bennett and V. Rajlich, "Software Maintenance and Evolution: A Roadmap," *22nd International Conference on Software Engineering (ICSE 2000)*. Limerick, Ireland, pp. 75–87, 2000.

[5] T. Mens, "The ERCIM Working Group on Software Evolution: the Past and the Future," *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*. ACM, 2009.

[6] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, and R. Hirschfeld, "Challenges in Software Evolution: Report of the ChaSE 2005 workshop organised by the ERCIM Working Group on Software Evolution," *IWPSE-05*. Lisbon, Portugal, pp. 13–22, 2005.

[7] R. Hirschfeld, K. Kawamura, and H. Berndt, "Dynamic Service Adaptation for Runtime System Extensions," *Wireless On-Demand Network Systems*. Springer Berlin Heidelberg, Madonna di Campiglio, Italy, pp. 227–240, 2004.

[8] B. P. Lientz and E. B. Swanson, *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations*. Addison-Wesley, 1980.

[9] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 5, pp. 309–332, 2003.

[10] B. Hardian, J. Indulska, and K. Henricksen, "Balancing Autonomy and User Control in Context-Aware Systems - a Survey," *Comorea, PerCom Workshops 2006*. IEEE, 2006.

[11] G. Biegel and V. Cahill, "A framework for developing mobile, context-aware applications," *the 2nd IEEE Conference on Pervasive Computing and Communication (PerCom)*, pp. 361–365 , 2004.

[12] T. Hofer, W. Schwinger, M. Pichler, G. Leonhartsberger, and J. Altmann, "Context-awareness on mobile devices – the hydrogen approach," *The 36th Annual Hawaii International Conference on System Sciences*, pp. 292–302, 2002.

[13] A. K. Dey, "Understanding and Using Context," *Personal Ubiquitous Computing*, vol. 5, no. 1, pp. 4–7, 2001.

[14] Q. Z. Sheng and B. Benatallah, "ContextUML: a UML-based modelling language for model-driven development of context-aware web services.," *Proceedings of the International Conference on Mobile Business (ICMB'05)*, pp. 206–212, 2005.

[15] K. Henricksen and J. Indulska, "A Software Engineering Framework for Context-Aware Pervasive Computing," *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications (PerCom 2004)*. IEEE, Orlando, FL, USA, pp. 77–86, 2004.

[16] M. Baldauf, S. Dustdar, and F. Rosenberg, "A Survey on Context-Aware Systems," *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 2, no. 4, pp. 263–277, 2007.

[17] J. Ye, L. Coyle, S. Dobson, and P. Nixon, "Ontology-based models in pervasive computing systems," *The Knowledge Engineering Review*, vol. 22, no. 4, pp. 315–347, 2007.

[18] H. Chen, T. Finin, and A. Joshi, "An ontology for context-aware pervasive computing environments," *Special Issue on Ontologies for Distributed Systems, Knowledge Engineering Review*, vol. 18, no. 3, pp. 197–207, 2004.

[19] C. Welty and D. L. McGuinness, "OWL Web Ontology Language Guide," vol. W3C Recomm. W3C Recommendation 10 Feb 2004, 2004.

[20] A. Shepherd, "HTA as a framework for task analysis," *ERGONOMICS*, vol. 41, pp. 1537–1552, 1998.

[21] E. Serral, P. Valderas, and V. Pelechano, “Towards the Model Driven Development of context-aware pervasive systems,” *Special Issue on Context Modelling, Reasoning and Management of the Pervasive and Mobile Computing (PMC) Journal*, vol. 6, no. 2, pp. 254–280, 2010.
[22] E. Serral, “Automating Routine Tasks in Smart Environments. A Context-aware Model-driven Approach,” Technical University of Valencia, 2011.
[23] S. J. Mellor and M. J. Balcer, *Executable UML: A Foundation for Model Driven Architecture*. 2002.
[24] J. Muñoz and D. V. P. Ferragud, “Model Driven Development of Pervasive Systems. Building a Software Factory,” Universidad Politécnica de Valencia, Valencia, 2008.
[25] M. B. Juric and P. Sarang, *Business Process Execution Language for Web Services: BPEL and BPEL4WS*. 2006.
[26] S. W. Loke, S. Smanchat, S. Ling, and M. Indrawan, “Formal Mirror Models: an Approach to Just-in-Time Reasoning for Device Ecologies,” *International Journal of Smart Home*, vol. 2, no. 1, pp. 15–32, 2008.
[27] “Code Generation conference - http://www.codegeneration.net/cg2010/,” 2010. .
[28] M. Guy, “Report 2: API Good Practice Good practice for provision of and consuming APIs,” UKOLN, 2009.
[29] J. Bloch, “How to Design a Good API and Why it Matters,” 2005, pp. 506–507.
[30] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, “Pellet: A practical OWL-DL reasoner,” *Journal of Web Semantics*, 2007.
[31] P. Bernstein, “Multiversion Concurrency Control - Theory and Algorithms,” *ACM Transactions on Database Systems*, vol. 8, no. 4, pp. 465–484, 1983.
[32] S. Cooper, W. Dann, and R. Pausch, “Alice: a 3-D tool for introductory programming concepts,” in *Journal of Computing Sciences in Colleges*, 2000, vol. 15, pp. 107–116.
[33] F. Pérez and P. Valderas, “Allowing End-users to Actively Participate within the Elicitation of Pervasive System Requirements through Immediate Visualization,” *Fourth International Workshop on Requirements Engineering Visualization (REV)*. IEEE, Atlanta, Georgia, USA, pp. 31–40, 2009.
[34] H. Lieberman, F. Paternó, and V. Wulf, *End User Development*. Springer, 2006.
[35] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.
[36] M. Van Welie and H. Trætteberg, “Interaction Patterns in User Interfaces.” pp. 13–16, 2000.
[37] Galitz and W. O., *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*. New York, NY, USA: John Wiley & Sons, Inc., 2002.
[38] B. Kitchenham, L. Pickard, and S. L. Pfleeger, “Case studies for method and tool evaluation,” *Software, IEEE*, vol. 12, no. 4, pp. 52–62, 1995.
[39] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, “Experimentation in Software Engineering,” *Springer*, 2012.
[40] J. V Jones, *Applied sofware measurement: assuring productivity & quality (2nd ed’97)*. McGraw-Hill, 1997.
[41] T. Strang and C. Linnhoff-Popien, “A context modeling survey,” in *First International Workshop on Advanced Context Modelling, Reasoning And Management at UbiComp 2004*, 2004.
[42] J. R. Lewis, “Psychometric Evaluation Of An After-Scenario Questionnaire For Computer Usability Studies : The ASQ,” *SIGCHI Bulletin*, 1991.
[43] D. J. Cook, M. Youngblood, I. I. I. E. O. Heierman, K. Gopalratnam, S. Rao, A. Litvin, and F. Khawaja, “MavHome: An agent-based smart home,” *In First IEEE International Conference on Pervasive Computing and Communications (PerCom’03),*. pp. 521–524, 2003.
[44] H. Hagras, V. Callaghan, M. Colley, G. Clarke, A. Pounds-Cornish, and H. Duman, “Creating an Ambient-Intelligence Environment Using Embedded Agents,” *IEEE Intelligent Systems*, vol. 19(6), pp. 12–20, 2004.
[45] P. Rashidi and D. J. Cook, “Keeping the Resident in the Loop: Adapting the Smart Home to the User,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 39, no. 5, pp. 949–959, 2009.
[46] G. I. Webb, M. J. Pazzani, and D. Billsus, “Machine learning for user modeling,” *User modeling and user-adapted interaction*, vol. 11, no. 1–2, pp. 19–29, 2001.
[47] L. G. Valiant, “A theory of the learnable,” *Communications of the ACM*, vol. 27, no. 11, pp. 1134–1142, 1984.
[48] E. Serral, P. Valderas, and V. Pelechano, “Improving the cold-start problem in user task automation by using models at runtime,” in *Information Systems Development*, 2011, vol. , 2011., pp. 671–683.

[49] M. García-Herranz, P. A. Haya, A. Esquivel, G. Montoro, and X. Alamán, "Easing the Smart Home: Semi-automatic Adaptation in Perceptive Environments," *Journal of Universal Computer Science*, vol. 14, no. 9, pp. 1529–1544, 2008.

[50] K. Henricksen, J. Indulska, and A. Rakotonirainy, "Using context and preferences to implement self-adapting pervasive computing applications," *Sofware: Practice and Experience*, vol. 36, no. 11–12, pp. 1307–1330, 2006.

[51] P. Johnson, "Tasks and situations: considerations for models and design principles in human computer interaction," *HCI International*. Munich, Germany, pp. 1199–1204, 1999.

[52] D. J. Cook and S. K. Das, *Smart environments: technologies, protocols, and applications*, vol. 43. Wiley-Interscience, 2005.

[53] F. Paternò, "ConcurTaskTrees: An Engineered Approach to Model-based Design of Interactive Systems," in *The Handbook of Analysis for Human-Computer Interaction*, 2002, pp. 483–500.

[54] C. Pribeanu, Q. Limbourg, and J. Vanderdonckt1, "Task Modelling for Context-Sensitive User Interfaces," *Interactive Systems: Design, Specification, and Verification (DSV-IS)*. Springer-Verlag Berlin Heidelberg 2001, Glasgow, Scotland, UK, pp. 49–68, 2001.

[55] N. Souchon, Q. Limbourg, and J. Vanderdonckt., "Task modelling in multiple contexts of use," in *Interactive Systems: Design, Specification, and Verification (DSV-IS)*, 2002, pp. 59–73.

[56] R. Huang, Q. Cao, J. Zhou, D. Sun, and Q. Su, "Context-Aware Active Task Discovery for Pervasive Computing," *International Conference on Computer Science and Software Engineering*. IEEE, Wuhan, China, pp. 463–466, 2008.

[57] J. P. Sousa, V. Poladian, D. Garlan, and B. Schmerl, "Task-based Adaptation for Ubiquitous Computing," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 36, 3, pp. 328–340, 2006.

[58] R. Masuoka, B. Parsia, and Y. Labrou, "Task Computing—The Semantic Web Meets Pervasive Computing," *2nd Int'l Semantic Web Conf on The Semantic Web (ISWC 2003)*, vol. LNCS 2870. Sanibel Island, FL, USA, pp. 866–881, 2003.

[59] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecturebased approach to self-adaptive software," *IEEE Intelligent Systems and Their Applications*, vol. 14(3) , pp. 54–62, 1999.

[60] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjørven, "Using Architecture Models for Runtime Adaptability," *IEEE Software*, 2006.

[61] B. Morin, J.-M. Jézéquel, F. Fleurey, and Arnor Solberg, "Models at Runtime to Support Dynamic Adaptation," *IEEE Computer Society*, pp. 46–53, 2009.

[62] C. Cetina, P. Giner, J. Fons, and V. Pelechano, "Using Feature Models for Developing Self-Configuring Smart Homes," *Fifth International Conference on Autonomic and Autonomous Systems*. IEEE, Valencia, Spain, pp. 179–188, 2009.

[63] D. Garlan and B. Schmerl, "Using Architectural Models at Runtime: Research Challenges," *Proceedings of the European Workshop on Software Architectures*. Springer Berlin Heidelberg, St Andrews, UK, pp. 200–205, 2004.

[64] M. Blumendorf, G. Lehmann, S. Feuerstack, and S. Albayrak, "Executable Models for Human-Computer Interaction," *In Interactive Systems. Design, Specification, and Verification Workshop (DSV-IS 2008)*. Springer Berlin Heidelberg, Kingston, Canada, pp. 238–251, 2008.

[65] C. Ballagny, N. Hameurlain, and F. Barbier, "MOCAS: a State-Based Component Model for Self-Adaptation," *Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE , San Francisco, California, pp. 206–215, 2009.

[66] M. Amoui, M. Derakhshanmanesh, J. Ebert, and L. Tahvildari, "Achieving Dynamic Adaptation via Management and Interpretation of Runtime Models," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2720–2737, 2012.

[67] G. Blair, N. Bencomo, and R. B. France, "Models@run.time," *IEEE Computer*, vol. 42, pp. 22–27, 2009.

[68] J. Zhang and B. H. C. Cheng, "Model Based Development of Dynamically Adaptive Software," *International Conference on Software Engineering (ICSE'06)*. ACM, Shanghai, China, pp. 371–380, 2006.