Dynamically Adapting BDI Agents Based on High-Level User Specifications

Ingrid Nunes[1,2], Michael Luck[2], Simone Diniz Junqueira Barbosa[1], Simon Miles[2], and Carlos J.P. de Lucena[1]

[1] PUC-Rio, Computer Science Department, LES - Rio de Janeiro, Brazil
{ionunes,simone,lucena}@inf.puc-rio.br
[2] King's College London, Strand, London, WC2R 2LS, United Kingdom
{michael.luck,simon.miles}@kcl.ac.uk

Abstract. Users are facing an increasing challenge of managing information and being available anytime anywhere, as the web exponentially grows. As a consequence, assisting them in their routine tasks has become a relevant issue to be addressed. In this paper, we introduce an adaptation mechanism that is responsible for dynamically adapting a BDI agent-based running system in order to support software customisation for users. This mechanism is used within a software framework for supporting the development of Personal Assistance Software (PAS), which relies on the idea of exposing a high-level user model to empower users to manage it as well as increase user trust in the task delegation process.

Keywords: Software Adaptation, BDI, Personal Assistance Software, Framework, User Modeling.

1 Introduction

Personal Assistance Software (PAS) is a family of systems whose goal is to assist users in their routine tasks. The popularity of these systems continues to increase with the increasing challenges faced by humans of being constantly accessible through mobile devices and dealing with huge amounts of information. Examples of this kind of system range from task managers, trip planners and recommender systems to robots that automate manual tasks. In this context, the development of PAS involves several challenges, mainly related to system customisation for individual users: user characteristics must be captured to provide personalised content and features for users, which involves eliciting, representing and reasoning about user preferences; and, once the optimal software configuration is determined based on these captured preferences, the system must adapt itself in order to provide personalised assistance for users.

Most current research concentrates on reasoning about user preferences, and identifying them by means of elicitation and learning processes. The application scenarios of such approaches personalise systems solely in terms of data, as in the case of recommender systems for instance. In contrast, in our work, we examine

F. Dechesne et al. (Eds.): AAMAS 2011 Workshops, LNAI 7068, pp. 139–163, 2011.

a different scenario: we aim to support the development of PAS systems that are able to modify their architecture dynamically to provide personalised functionality as well. Customisations are specified directly by users, by choosing from the *services* provided and customising them with optional features and preferences, so that users can understand the system evolution (giving *transparency*), and control the adaptation process (giving *power of control*). Our ultimate aim is to build software to automate user tasks; the ability to understand what the system knows about users and providing them with a means for controlling the system are key issues to be addressed.

In this paper, we propose an adaptation mechanism for dynamically adapting a running belief-desire-intention (BDI) agent-based system, as a means of supporting software customisation for users. The mechanism consists of a generic process that produces adaptation actions to modify the state of a running system based on changes to a user model expressed in terms of a vocabulary that is understood by end-users, so that the end-user and implementation levels are always in a consistent state. This generic process is instantiated with the abstractions of a PAS Domain-Specific Model (DSM) [17] and the BDI architecture [20]. The widely adopted BDI architecture provides abstractions, and a reasoning mechanism, suitable for developing cognitive agents, in particular agents able to automate user tasks. Moreover, the architecture is composed of loosely coupled components and makes an explicit separation between what to do (goals) and how to do it (plans), making it very flexible, and facilitating the adaptation process by evolving and changing components with a lower impact on the running system.

We also introduce a software framework that is built based on the proposed PAS DSM and adaptation mechanism, in order to provide a large-scale reusable infrastructure for the development of PAS. Our framework not only incorporates the idea of having two abstraction levels that are kept consistent by dynamic adaptation, but also addresses other problems that are common to PAS systems, including the proposal of an architecture for this kind of system. We evaluate our adaptation mechanism and framework with a qualitative analysis, by discussing design decisions, software quality attributes and drawbacks.

The remainder of this paper is organised as follows. We begin by introducing the PAS DSM in Section 2, which drives the adaptation process. Sections 3 and 4 describe the dynamic adaptation mechanism and the PAS framework, respectively. Section 5 provides qualitative analysis of our proposal by discussing relevant aspects from it. Finally, Section 6 discusses related work, followed by Section 7, which concludes.

2 PAS Domain-Specific Model

2.1 Overview

The PAS Domain-Specific Model (DSM) has a key role in our approach, as it drives adaptations in PAS systems. In short, it is a meta-model that defines

abstractions for modelling domain-specific concepts of PAS, such as features and preferences, with the goal of using abstractions closely related to the user's vocabulary. The PAS DSM presented in this paper is an updated and revised version of the model defined in our previous work [17]; some previously described parts are only briefly introduced, and we refer the reader to the provided reference for further details.

The central concept of the PAS DSM is the *user model*, which captures individual *customisations* at the user level in a high-level language. These customisations can be *configurations* or *preferences*. *Configurations* are direct and determinant interventions that users perform in a system, such as adding or removing services and enabling optional features, and which can be related to environment restrictions (e.g. a device configuration, or functionalities provided by the system). *Preferences* provide information about user values that influence decision making, and thus can be used as resources in agent reasoning processes; they typically indicate how a user rates certain options better than others in certain contexts.

In order to build user models, we need a set of *definition models* that define the abstractions of a PAS that characterise a particular application. These abstractions in definition models provide both domain entities (such as features and ontology concepts) to be referred to in user models, and restrictions used for defining valid user models. Definition models and user models are instantiated in a stepwise fashion, as follows. The former is constructed by system developers during the instantiation of a PAS application; that is, it is a consequence of *design decisions*. The latter is instantiated at runtime by users (possibly by the system learning from users), so that it will correspond to *user decisions*. More specifically, there are three different definition models, the *ontology model*, the *feature model* and the *preference definition model*, which we consider in more detail below.

Before proceeding to detail these models, however, and in order to make it easier to understand our models and later our adaptation mechanism, we use the scenario of a *smart home* as illustration. A smart home is a software system that controls a robot, whose aim is to assist individuals to do their housework by means of its various capabilities, which allow it to undertake many tasks, such as cooking, cleaning, and washing clothes. Users can customise the robot, first by choosing from these available tasks and second by tailoring them according to their preferences. For example, some homes are cleaned with a vacuum cleaner while others are cleaned with a broom, and users can choose between them, as indicated above, while at the same time, users can influence the order of cleaning the rooms in the house in line with their preferences. Similarly, in order to choose a dish to be cooked, the system must be aware of the user's food preferences. Finally, different users may want to give different levels of automation for the system; for example, the system might be responsible for suggesting a particular dish and cooking it, but a particular user, who wants to maintain a higher level of control, may choose to decide or approve what to cook.

2.2 Ontology Model

The *ontology model* defines the set of concepts within the application domain and the relationships between them. Concepts represented in the ontology model are used in other models; for instance, we may need to define the elements of the domain, such as the rooms in a house, which are then used as part of the preferences expressed in the user model itself. There are four different kinds of elements in the ontology model, as follows.

- *Classes* are coarse-grained concepts of the ontology; for instance, *Dish*, *Food* and *Quantity*. *Food* is anything that can be eaten, *Dish* is a type of *Food*, which requires preparation, and *Quantity* expresses an amount of something.
- *Properties* can be associated with these classes, where each property has a range of values (or its *domain*) that can be assigned to the property. When a class is instantiated, each property takes a value that is part of its domain. For example, *Quantity* has two properties: *amount*, whose domain is a real number, and *unit*, whose domain is the enumeration *Unit* (see next item below). *Dish* has a property named *ingredients*, whose domain is a mapping from *Food* to *Quantity*, indicating how much of the instances of *Food*, such as sugar and milk, is used; for example, three (*amount*) spoons (*unit*). Classes are coarse-grained as they have self-contained meaning as opposed to their properties, which are fine-grained as they depend on the association with a class to have a meaning.
- *Enumerations* are particular kinds of domain, which are composed of sets of *named* values (*enumeration values*). For instance, the enumeration *Unit* is composed of the enumeration values, *teaspoon*, *tablespoon*, *cup*, and so on.
- *Value domains* can be seen as particular kinds of enumeration, being composed of *values*. In this context, this notion of *value* [15] describes preferences not over the characteristics of a concept but over the value it promotes, and is a first-class abstraction that we use to model high-level user preferences. For example, in the value domain of *food*, a value can be *health*, where a salad promotes health.

2.3 Feature Model

The *feature model* defines the set of services or capabilities that can be configured in PAS applications by users; we refer to these generically as features, and they can be any characteristic relevant to the user. For instance, a feature might be a functionality or a setting, such as the presence of the washing clothes functionality. More specifically, our feature model is an extended and adapted version of the feature models used in the Software Product Line (SPL) context [8]. SPL is a new software reuse approach that aims at systematically deriving families of applications based on a reusable infrastructure in order to achieve both reduced costs and time-to-market. A PAS application can thus be seen as a SPL whose products are applications customised for a particular user.

Mandatory features are associated with invariant parts of the application, which can be customised with optional or alternative (sub-)features. *Optional*

and *alternative* features correspond to parts of the application that can vary for different instances, where the latter features must also respect the cardinality of the feature *group* they belong to. Here, feature groups allow alternative features to be grouped together, and have an associated cardinality, indicating the minimum and maximum number of alternative features that can be selected in a *configuration* (as part of the user model).

Now, a key characteristic of PAS is providing users with functionalities that automate their tasks, but feature models from SPL do not explicitly capture this aspect. Therefore, we enrich these feature models by distinguishing a particular kind of feature, *autonomous features*, which provide the functionality of *acting on behalf of users for performing a user task*. Each autonomous feature is associated with a set of autonomy degrees, corresponding to the level of autonomy available to the user — initiate (`I`), suggest (`S`), decide (`D`) and execute (`E`) — according to the taxonomy presented for adaptive systems [16]. The availability of these autonomy degrees does not imply that any particular degree will be adopted; instead, this is determined in the *configuration* part of the user model.

Finally, we need to consider how we can restrict the combination of features to allow the selection of only valid sets. To do so, we use configuration constraints on features, classical logic expressions whose evaluations determine if the set of selected features is valid.

Given these considerations, our feature model consists of a tree and a set of constraints, where the tree nodes are (mandatory, optional, or alternative) features and feature groups, except for the root node, which must be a mandatory feature. Each feature has an identifier, a set of *autonomy degrees* and, as children, a set of (mandatory or optional) features and a set of feature groups. In turn, feature groups have a set of alternative features and their associated cardinality in addition to an identifier.

Figure 1 shows the feature model of the smart home example, using the notation based on [14], with extensions. It shows the features that this PAS offers, and the different configurations it can have; for instance, the *cleaning* feature has two options, *vacuum cleaner* or *sweep*. In addition, degrees of autonomy are associated with features. All degrees are offered to users, who choose from them for the system with respect to the *cooking* feature: (i) `I` — the robot *initiates* the process of cooking or performs cooking only when the user requests it; (ii) `S` — the robot chooses a set of possible options of dishes to be cooked or the user provides these options; (iii) `D` — the robot chooses an option to be cooked, or the user makes this decision; and (iv) `E` — the robot cooks the chosen dish or the user cooks it.

2.4 Preference Definition Model

It is desirable that users are able to express preferences in different ways, but this must be restricted to those that can be understood by the system. For instance, if an application can deal only with *quantitative* preferences, *qualitative* preferences will have no effect on system behaviour if there is no mechanism to translate them to quantitative statements. The Preference Definition Model (PDM) thus

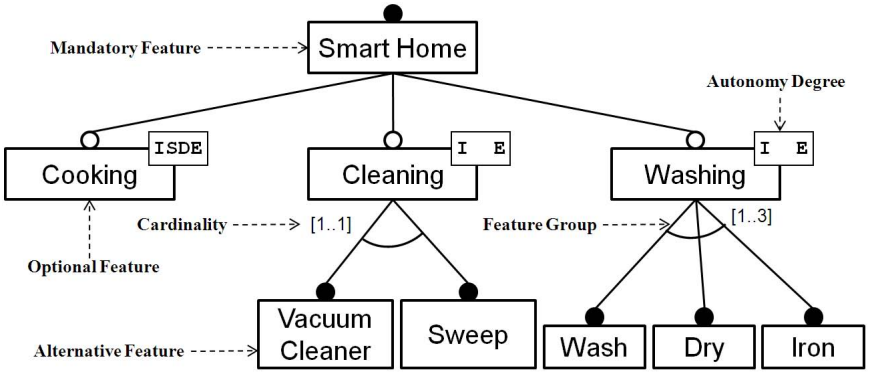

Fig. 1. Smart Home Feature Model

specifies restrictions over the expression of user preferences in that its purpose is to constrain *how* users can express preferences and *about which elements* of the ontology model.

Our consideration of the different kinds of preference statements available is informed by user studies in which actual preference statements were collected from different individuals and from existing preference reasoning models. The aim is to maximise the expressiveness available to users, and leads us to five distinct kinds, as follows.

Order. Order statements establish an order relation between two elements, stating that one element is preferred (strictly or not) to another. A set of instances of order preferences comprises a partial order. Example: *I prefer* $< target_1 >$ *to* $< target_2 >$.

Rating. In rating statements, users attribute a *rate* to a target, where this rate must belong to a *rating domain* that is associated with that target. Besides implicitly establishing an order relation among elements, a rating preference also indicates how much an element is preferred (or equivalent) to another, with respect to a given feature. Example: *I rate* $< target >$ *with the value* $< rate >$.

Maximisation/Minimisation. Statements that indicate that the user preference is to minimise or maximise a certain element. Example: *I prefer to maximise* <*target*>.

Reference Value. The reference value preference enables users to indicate one or more preferred values, or a range of values, for an element. Example: *I prefer* <*target*> *as close as possible to* $< reference\ value >$.

Don't Care. Don't care statements of this kind indicate a set of elements that the user does not care about, they are equally (un)important to the user. Example: *I am indifferent to* $< target_1 > ... < target_n >$.

Preference targets can be associated with subsets of these kinds of preference statements, so that it is possible to express only such preferences about the target. In addition, in the particular case of rating preferences, we also need rating

domains comprising specific value statements or *rates*. Such domains can be numeric (either continuous or discrete), with specified upper and lower bounds, or enumerations such as { *love*, *like*, *indifferent*, *dislike*, *hate* }.

Targets can be one of four types, as specified in the ontology model (class, property, enumeration, and value domain). By default, when no preferences are defined for a target, any kind of preference statement can be made about it, including rating preferences with their defined rating domains. Thus, for example, if the class *Dish* is associated with an *order* preference, order preference statements about its enumeration values can be provided.

2.5 User Model

Given these various definition models, we can proceed to considering the user model, which specifies user customisations for each individual. This user model is constructed with abstractions from these definition models, which also constrain it. As stated previously, customisations can be either configurations or preferences, both of which are needed in the user model. The configuration comprises: a set of features selected from, and valid according to, the feature model; and a set of feature autonomy configurations that state the autonomy *degree* specified for an autonomous feature (a feature whose autonomy degree set is not empty).

Figure 2 shows how the previously presented models are related, and how user models are based on them. The feature model provides the features available for users, and autonomy degrees of autonomous features, while the ontology model provides users with a vocabulary to construct preference statements. Based on these two models, users create or *evolve* a user model, which is then validated according to the feature model (that also contains constraints over selected features) and the PDM. A set of selected features is valid according to the feature model when: (i) it contains all mandatory features; a subset (which can be empty) of the optional features; and a subset (which can be empty) of the alternative features; (ii) selected alternative features must respect the cardinality of the feature group they belong to; (iii) all constraints of the feature model must be evaluated for a true value; and (iv) optional and alternative features can be selected only if all their parents are selected. A preference statement is valid if its targets have the associated allowed preference specified in the PDM, or if there is not definition in the PDM, since the default is that all preference types are allowed. In the smart home example, an example of a user model is as follows.

(i) $Configuration$
- $Features = \{Cooking, Cleaning, Sweep\}$,
- $AutonomyDegree = \{Cooking \mapsto \{S, E\}, Cleaning \mapsto \{I, E\}\}$

(ii) $Preferences = \{\text{“}I\ prefer\ pasta\ to\ meat''\text{, “}Maximise\ health''\}$

We have now described the set of models that are part of the PAS DSM. Definition models together provide a language — domain entities, features, preference

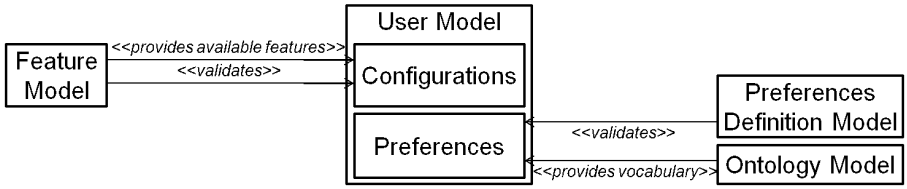

Fig. 2. Relationships among models

statements — for the the instantiation of user models, which allow customisations of systems to be captured by using an application-specific vocabulary that is close to the end-user language, so that users can understand and manage these models. Now, the abstractions used to build user models are related to pieces of software assets that implement them, which are instantiated and composed according to a particular user model state. Moreover, as user models can be changed at runtime, this instantiation and composition process must be performed dynamically, and this process of dynamically adapting running systems, in particular based on BDI agents, is presented next.

3 Dynamically Adapting BDI Agents

As described in the introduction, our approach requires synchronisation between the user and implementation levels, because the former represents user customisations at a high-level (user model) and the latter must reflect these customisations. Importantly, we allow the user to change these customisations at any point while the system is running, requiring the implementation also to change. This is achieved by an adaptation process, that is elaborated in this section, first in a generic fashion, independent of the underlying architecture, and then extended for our particular user model and the BDI architecture.

3.1 A Generic Mechanism to Support Dynamic Adaptation

Our adaptation mechanism is triggered by changes (which we assume are made by the user) to the user model, so that when the user model reaches a new state, the adaptation mechanism causes the running application, in our case the implementation level of the PAS, to be consistent with it. In a nutshell, the modification of the user model state is achieved in relation to a set of *events*, which cause the user model to change from a state um to a state um'. The running system monitors events, and when their occurrence is detected, a set of adaptation *rules*, which are associated with at least with one of these events, is invoked to generate adaptation actions. These *actions* make changes to the running application, typically by adding and removing software assets that are either coarse-grained or fine-grained parts of the running system, such as

agents and beliefs. Before proceeding to provide an explanation of how our mechanism can be applied to the BDI architecture, we introduce each of these concepts.

Events. Events correspond to changes that occur in the user model; for instance adding or removing the autonomy degree of a feature. We denote the set of events by E, i.e. $E = \{e_1, ..., e_n\}$, where each e_i is an event.

Event Categories. Events can be grouped according to a certain criterion, to form event categories. For example, the events above belong to the autonomy degree category. We denote the set of event categories by EC, and each $ec_i \in EC$ consists of a subset of events; that is, $ec_i = \{e_1, ..., e_k\}$, where each $e_i \in E$. Events can be associated with more than one category.

Adaptation Actions. Actions are the changes to be performed over software assets within the running application, for example by adding or removing agents, beliefs or goals. The set of actions is referred to as $A = \{a_1, ..., a_n\}$, where each a_i is an action, and $a_i = SS \rightarrow SS$. SS denotes the set of system states, so that an action changes the running system from a state ss to a state ss'.

Adaptation Rules. Rules establish connections between events and actions, and are applied when an event of some category associated with the rule occurs. In this situation, the rule generates the appropriate set of actions to be executed, according to the event(s) that occurred. It is important to highlight that rules are not functions only in terms of events, in the sense that the same set of events does not always generate the same (and unique) set of actions, because generated actions may depend on the previous state of the user model. Therefore, rules are defined as $R = \{r_1, ..., r_n\}$, and each rule $r_i = \langle \{ec_i\}, UM \times UM \times \{e_i\} \rightarrow \{a_i\} \rangle$, where UM is the set of states (or instances) of the user model.

For instance, let r_1 be a rule stating that agent X must be part of the running system if features $F1$ and $F2$ are selected. Then, if the event $Feature(F1, add)$ occurs, r_1 generates the action $a_1 = Agent(X, add)$ only if feature $F2$ was previously selected or the event $Feature(F2, add)$ also occurs.

These concepts comprise the knowledge that captures how a running system must be adapted in line with changes to the user model. More specifically, the adaptation mechanism, illustrated in Figure 3, takes as input a previous and an updated version of a user model and adapts the system as follows. First, the set of all events that caused the user model to be updated is generated. Then, the set of rules that are triggered by at least one of the categories from those events is selected. Next, a set of actions to be executed is constructed from the union of each set of actions generated by each selected rule. Finally, each action is executed, changing the system state from ss to ss', consistent with the updated version of the user model. Now, in order to make this applicable to a particular architecture, we must instantiate the adaptation mechanism so that it includes a specification of the initial state of the user model, which is used in the first run of the system.

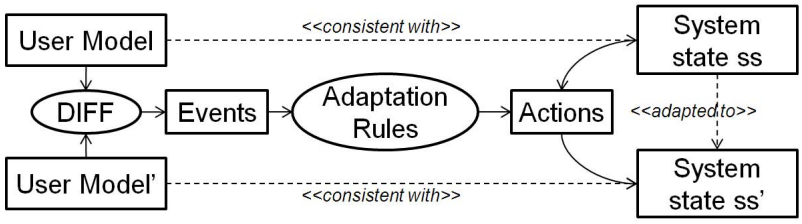

Fig. 3. Adaptation Process

3.2 Extending Our Mechanism to the BDI Architecture

It can be seen that the generic adaptation mechanism is simple in that it does not specify any particular events, categories, rules or actions, so that it can be extended to any particular instances of these concepts. Indeed, we have done just this, and have extended the mechanism for our particular user model, as presented in Section 2, and the BDI architecture, which we adopt to support the development of PAS.

In our approach, agents follow the BDI architecture [20] mainly because it is a flexible architecture that separates what must be done (agent goals) and the course of actions (plans) that should be executed to achieve goals. As our focus is to customise systems for users, and this explicit separation is an important property that helps to support customisation. BDI agents are composed of beliefs (the agent's current view of the world, composed of a name and a value or a set of values), goals (the desires it wants to achieve) and plans (pre-defined course of actions to achieve goals). These abstractions are part of the BDI architecture, and we use an extension of it, which includes capabilities [7] that are modularised reusable parts of agents, which consist of a set of goals related to the capabilities, plans for achieving these goals and the necessary set of beliefs (knowledge) needed for executing these plans. In our approach, an agent has a set of goals and a set of capabilities. Capabilities have a set of beliefs (belief base) and a set of plans (plan library).

In what follows, we describe the instantiation of the presented adaptation mechanism with specific events, event categories, actions and rules that are required for this instantiation. We also introduce the concept of *event type*, which can be *add* or *remove* (i.e. $EventType = \{add, remove\}$), as it is relevant to many of the kinds of operations we require.

Events. There are six different kinds of events associated with our user model, which are represented by three entities, each associated with an event type, as follows.

- $FeatureEvent(f, et)$, where f is a feature and et is an event type, indicating that f was added or removed.
- $AutonomyDegreeEvent(f, ad, et)$, where f is a feature, ad is an autonomy degree and et is an event type, indicating that ad of f was added or removed.

– $PreferenceEvent(p, et)$, where p is a preference and et is an event type, indicating that p was added or removed.

As stated above, there must be an *initial* specification of the state of the user model in order to bootstrap the adaptation mechanism. In our case, the initial state is a configuration with the core (mandatory) features selected and no preference statements. As a result, changes (events) are detected using this configuration of the user model as a baseline when the user model is first updated.

Event Categories. There are eight different types of event categories, which are enumerated next. For event categories that are related to preferences, we use our running example.

– $FeatureEventCategory(feature)$, consisting of events of adding or removing feature $feature$.
– $AutonomyDegreeEventCategory(feature, autonomyDegree)$, consisting of events of adding or removing the autonomy degree $autonomyDegree$ of feature $feature$.
– $EnumValuePreferenceEventCategory(enumValue)$, consisting of events of adding or removing a preference that refers to an enumeration value $enumValue$, such as the meal types *lunch*, *snack* and *dinner*.
– $EnumPreferenceEventCategory(enum)$, consisting of events of adding or removing a preference that refers to any enumeration value of the enumeration $enum$, such as the enumeration *meal type*.
– $ValuePreferenceEventCategory(value)$, consisting of events of adding or removing a preference that refers to the value $value$, such as $Health$.
– $ClassPreferenceEventCategory(class)$, consisting of events of adding or removing a preference that refers to the class $class$, such as $Dish$.
– $PropertyPreferenceEventCategory(property)$, consisting of events to add or remove a preference that refers to the property $property$, such as *meal type* and *ingredients*.
– $EntityPreferenceEventCategory(class)$, consisting of events of adding or removing a preference that refers to the class $class$ or any of its properties.

Adaptation Actions. Adaptation actions are responsible for changing the running system state, and in our instantiation of the adaptation mechanism for BDI architectures, they add or remove software assets, where a *software asset* is any part of the implemented system. Fine-grained assets are those that are not self-contained, like beliefs, plans and goals. Coarse-grained assets are either *components* and *agents*. The former provide reactive behaviour, while the latter provide autonomy and pro-activity, have their own thread of execution, and are able to communicate through messages with other agents.

We specify nine types actions to manipulate software assets of the running system, parameterised with the asset being manipulated and an action operator $AO = \{add, remove\}$, as follows.

- $AgentAction(agent, ao)$ starts the agent *agent* when *ao* is *add*, or kills it when *ao* is *remove*.
- $CapabilityAction(agent, capability, ao)$ adds or removes the specific capability *capability* from the agent *agent*, according to the given action operator.
- $BeliefAction(capability, belief, ao)$ adds or removes the belief $belief$ from the belief base of the capability *capability*, according to the given action operator.
- $BeliefValueAction(capability, belief, object)$ performs a belief update, by setting the value *object* of the belief of the capability, whose name is given in $belief$ (we use the term *object* to avoid referring to the term *value* as in the ontology model).
- $BeliefSetValueAction(capability, belief, object, ao)$ adds or removes the object *object* from the set of values, according to the given action operator, where belief $belief$ is associated with a set of values, and must be part of the capability *capability*.
- $GoalAction(agent, goal, ao)$ adds or removes the goal *goal* from the agent *agent*, according to the given action operator.
- $PlanAction(capability, plan, ao)$ adds or removes the plan *plan* from the capability *capability*, according to the given action operator.
- $ComponentValueAction(female, interface, male)$ plugs the *male* component into the given $interface$ of the $female$ component, and consequently removes a previous component, if it was plugged to this $interface$. If no component is given for the *male* component as a parameter, this action will unplug the component that is currently connected to the interface of the $female$ component.
- $ComponentAction(female, interface, male, ao)$ has similar behaviour to the action above, as it (un)plugs *male* components from the $interface$ of a $female$ component. In this case, the $female$ component has an $interface$ with many entrances, and therefore it allows many components to be plugged into it. When the action operator is *add*, the *male* component is plugged to one of the entrances of the $interface$; when the action operator is *remove*, it removes it.

Adaptation Rules. We have developed three types of adaptation rules, which follow the same pattern and are related to features and autonomy degrees. The pattern we have identified is the analysis of the presence of a characteristic in the user model, such as the selection of a feature. If this characteristic was *not* present in the previous version of the user model, but it is present in the updated version of the user model, then the rule generates a set of adaptation actions composed of two parts: (i) *onActions* (or *onA*) are actions for which the action operator is set to *add*, or for which the *male* component has a given value (plugging in the component), and (ii) $offActions$ (or $offA$) are actions for which the action operator is set to *remove*, or the *male* component has a *null* value (unplugging the component). If the characteristic *was* present in the previous version of the user model, but is not present in the updated version of the user model, the sets *onA* and $offA$ have the operators set in the opposite way. Finally, if

there is no change to the presence of the characteristic, no adaptation action is generated. Thus, rules that follow this pattern must provide the characteristic to be analysed together with prototypes of actions of the sets onA and $offA$. The three provided rules are as follows.

- $FeatureExpressionRule(featureExpression, onA, offA)$: the presence condition is a logic formula with the connectors *not*, *and* and *or*. Literals represent features in a configuration, and evaluate to *true* when the feature is *selected* in the configuration. The event categories associated with this rule correspond to the instances of $FeatureEventCategory$ identified by the literals within the $featureExpression$.
- $OptionalFeatureRule(feature, onA, offA)$: this is a particular case of the previous rule, whose expression is only the literal $feature$.
- $AutonomyDegreeRule(autonomyDegreeEventCategory, onA, offA)$: the presence condition is the selection of the autonomy degree of the feature specified in the $autonomyDegreeEventCategory$, an autonomy degree event category given as parameter, which is the event category associated with this rule.

We illustrate an adaptation by showing a rule related to the degree of autonomy *decide* of the *cooking* feature. When this degree of autonomy is given, the system — and in this case the *robot* (an agent) with the *cooking* capability — must execute a plan to automatically choose a dish from a given set of options. Alternatively, the system must interact with the user directly and request them to make this choice, which comprises another plan. Both plans achieve the *decide cooking* goal, but only one of them should be part of the plan library of the *cooking* capability, and this selection is made according to the degree of autonomy given to the system. Note that the user is only aware of features, degrees of autonomy and preferences, while software assets are transparent to them; rules are responsible for connecting these two levels. We show the example rule below, which we refer to as *DecideCookingRule*.

DecideCookingRule = $AutonomyDegreeRule($
$AutonomyDegreeEventCategory($**CookingFeature**, **Decide**$)$,
$\{PlanAction($**CookingCapability**, **ChooseDishPlan**, **add**$)\}$,
$\{PlanAction($**CookingCapability**, **AskUserToChooseDishPlan**, **remove**$)\}$$)$

We remind the reader that the action operator parameter of actions is irrelevant in the rule definition. When the user model is updated and the *decide* degree of autonomy of the *cooking* feature changes from selected to unselected, this rule is executed. Thus, the set of actions onA is instantiated with the action

operator *add*, and the $offA$ set is instantiated with the operator off – and actions of both sets are executed. If the opposite change is made (unselected to selected), the operators are interchanged. Finally, if no change is made, this rule is not executed, and there is no inclusion or exclusion of those plans.

There are no pre-defined rules related to preferences because our approach is independent of the mechanism adopted for reasoning about preferences, which is tailored to specific applications (such as through utility functions, for example) and should thus have separate rules. In order to illustrate how rules can be applied for preferences, we also use the *cooking* feature as example. Assume that that the $ChooseDishPlan$ implements a domain-neutral approach for reasoning about preferences, namely *CP-nets* [5]. This plan implements the algorithm proposed in this approach for choosing a food option, and uses a belief named $foodCPnet$, which represents user preferences using the CP-net structure. In this case, a rule is created with the following characteristics: (i) it is associated with the $EntityPreferenceEventCategory$ parameterised with the $Food$ class, and therefore will be triggered by the addition or removal of any preference about food; and (ii) when a preference is added or removed, the rule produces an updated CP-net and changes the value of the $foodCPnet$ belief.

4 A Two-Level Framework for Developing PAS

We have now described the PAS DSM, which provides means for building high-level user models, and the adaptation mechanism that is driven by changes in these user models and allows specifying rules for dynamically adapting a running system based on agents following the BDI architecture. In this section, we show how these two previously introduced components are used together to structure and build a software framework to support the development of PAS.

A software framework is a reusable infrastructure — typically code and associated documentation showing how to extend it — which abstracts application-specific details, provides generic functionality to facilitate implementation, and promotes large-scale reuse, higher quality and increased speed in the development process. A key difference from libraries is that frameworks control the flow of the running system, and invoke extensions constructed for specific applications, a characteristic that is referred to as *inversion of control*.

Our framework is thus a domain-neutral reusable software infrastructure for developing a family of systems to assist users in their routine tasks in a customised way. It is aimed at supporting development of agent-based PAS, whose variability is expressed in terms of features, and which takes into account user preferences to provide customised behaviour. In addition to the smart home, we can also consider the example of a system to manage a car. Modules that automate tasks, such as breaking, changing gears and controlling the radio, can be customised through configurations (that are enabled modules with respective settings) and preferences (the driving-style and music preferences).

The main characteristic of our approach is the adoption of two levels of abstraction that capture user customisations: the (end-)user and implementation

levels. The user level enables customisations to be explicit and modular, as well as being understandable by users, so that the current state of the PAS is transparent to users, and empowers them to manage customisations. As users evolve and personalise their PAS over time, and since there is an underlying implementation that must be consistent with the high-level user specifications, there is a need to keep both levels synchronised. Thus, changes at the user level drive dynamic adaptations in the underlying implementation, in order for the latter to reach a state consistent with the user level representation. In this section, we provide details of our framework and then briefly describe the steps needed to instantiate it for specific application domains.

4.1 Framework Architecture and Details

Our framework supports the implementation of PAS using a client-server model, which allows the processing to be distributed among individual clients, and at the same time provides mobility for users. As illustrated in Figure 4, clients and server have different responsibilities and different software architectures but, since they must communicate with each other, they share a common module composed of *definition models*, which contains domain abstractions (the ontology) and message templates. The figure shows this common module between the client and server to indicate that it is the same for both sides. *Definition Models* of the framework contain only the domain-neutral abstractions related to PAS, but can be extended to include domain-specific abstractions when instantiating applications.

User data is stored in a centralised database on the server, which is structured in layers, with a *Business* layer that provides services for PAS clients, and a *Data* layer composed of Data Access Objects (DAOs) [1] that access the database. Both can be extended to incorporate application-specific services. PAS clients, in turn, have the following components.

***Configuration* module.** This module enables users to manage the user model, which represents customisations at a high level. It provides the functionality of building a graphical interface for users to manipulate the user model, by

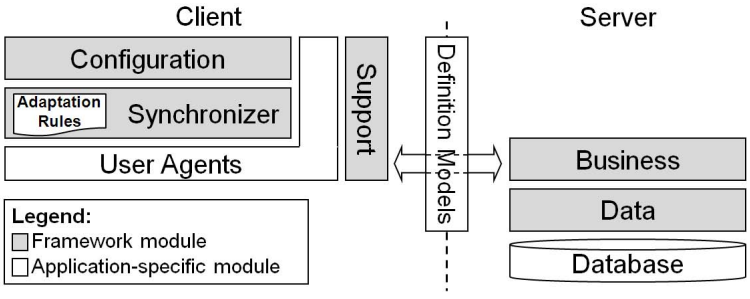

Fig. 4. The PAS Client-server Architecture

retrieving definition models from the server. It also validates user changes to the model, and stores new versions in the server, triggering the system's adaptation process.

***User Agents* module.** This module is application-specific, and it provides application-specific services and functionalities for users. Since variability (i.e., service parts that can be customised for users) must be taken into account when developing such servers, the implementation of PAS can be seen not only as a unique software system but as a set of software assets that can be integrated to form different customised applications for diverse users. Thus, all optional parts of the PAS must be modular in the code so that they can be added and removed from the running application instance. Each set of assets that realises a variable part of the PAS is the implementation level representation of user customisations.

***Synchronizer* module.** This module implements and executes the adaptation process presented in Section 3 to keep the previous two modules consistent, when the user model is modified. It realises the generic adaptation process, and also provides the specific events, categories, actions and rules. Application-specific parts of this process consist of the declaration of adaptation rules and their parameters, and instantiations of other kinds of rules, for example for dealing with preference statements.

***Support* module.** There are many services that are common to the family of PAS, so in this module we provide core PAS services, such as login.

In Figure 4, in which we show the modules of the architecture, we highlight those that are provided by the framework and those that should be implemented in specific applications.

We adopted an agent-based approach for implementing the framework, and one can observe the multi-agent interaction from macro and micro-level views, as shown in Figure 5. From a macro-level viewpoint, PAS clients can be seen as autonomous and proactive agents that represent users in a multi-agent system. These PAS clients communicate with the PAS server to access stored information and other business services, and can also communicate with each other as well as with services available on the web. For example, if our framework is instantiated for the *trip planning* domain, services are agents representing hotel and airline companies. A PAS client at the macro-level can be seen as a single agent representing a user, but at the micro-level it is decomposed into autonomous components (also agents), each of which has different responsibilities.

In order to support the development of BDI agents, we had to choose a BDI agent platform, as the implementation of agents involves many concerns, such as thread control and message exchange across the network, which are costly if implemented from scratch. While several agent platforms implement the BDI architecture (e.g., Jason, Jadex, 3APL and Jack), and most are based on Java, the programming language adopted in our framework, the agents are implemented in these platforms using a particular language that is later compiled or interpreted by the platform. This prevents us from taking advantage of the Java language features, such as reflection and annotations, which can help with the

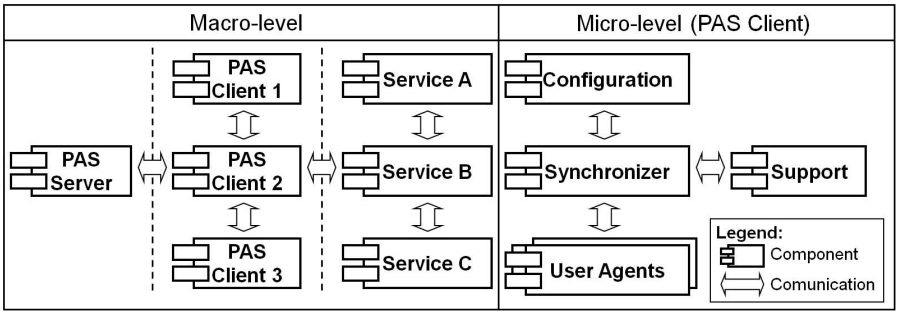

Fig. 5. Macro and Micro-level Views

implementation of our adaptation mechanism, and complicates integration with other frameworks, in particular Spring. For example, existing BDI platforms have declarations of agents in a particular file and, based on these declarations, the platform creates instances of agents, beliefs, plans, and so on, and connects them. In addition, it is hard to obtain references for these created instances and manipulate them. This is the main difficulty we found in implementing our adaptation mechanism — we need a repository of components, so that we can (un)plug them, and existing BDI platforms do not provide a means of doing this.

Due to this limitation of existing BDI platforms, we have developed BDI4JADE,[1] a BDI layer on top of JADE[2] (a Java-based agent platform that provides a robust infrastructure for implementing agents, including behaviour scheduling, communication and a yellow pages service). Since BDI4JADE components are extensions of Java classes, they can be instantiated by other frameworks and plugged into the running application, as opposed to other agent platforms that instantiate and manage their components. Further details of BDI4JADE is published elsewhere [18].

Our adaptation mechanism was implemented with extensive use of the Spring framework,[3] a Java platform that provides comprehensive infrastructure support for developing Java applications. It is designed to be non-intrusive, so that the domain logic code generally has no dependencies on the framework itself. Mostly, we took advantage of the Dependency Injection and Inversion of Control module, which allows declaration of the application software assets (*beans*, in Spring terminology) and dependencies among them. Thus, actions in our adaptation mechanism receive strings (bean identifiers) as parameters, referring to software assets of the running PAS to be adapted. These bean declarations can correspond to either a singleton or a prototype instance of the bean.

In summary, the implementation of the adaptation mechanism contains a set of classes of all concepts introduced in this paper (all models of the PAS DSM and concepts of the adaptation mechanism), as well as agents presented

[1] http://www.inf.puc-rio.br/~ionunes/bdi4jade
[2] http://jade.tilab.com/
[3] http://www.springsource.org/

in this section. A Spring configuration file has the declaration of software assets (agents, beliefs, plans and so on), and therefore we have a repository of prototypes or singleton components. Rules and actions are also declared in this configuration file, making reference to software assets, and specifying how components should be (un)plugged. The `Synchronizer` agent (on the right side of Figure 5) loads all rules on its initialisation, and runs the adaptation process when the `Configuration` agent notifies it of a change to the user model. The `Synchronizer` agent loads the software asset instances and invokes methods by reflection to connect the components at runtime.

4.2 Instantiation

In order to instantiate our framework for specific applications, the modules indicated in Figure 4 need to be developed. First, the ontology model must be extended to incorporate domain-specific abstractions; for example, in a *trip planning* system, this involves travel-related entities and messages to be exchanged with hotels and flight company agents. Second, the other definition models must be specified, in order to build user models. Third, since user models persist in the server database, a database must be chosen and set up in the system. We use a Java persistence framework, so it is only necessary to indicate the database used in a configuration file, and to provide the object relational mapping of application-specific abstractions that must persist.

Once abstractions and models are specified, application-specific services must be developed, by implementing user agents by means of techniques to support the variability specified in the definition models. Finally, adaptation rules must be specified to model the knowledge necessary for the application to change as the user model is updated.

5 Discussion: A Qualitative Analysis of Our Approach

As an initial step for the evaluation of our adaptation mechanism and framework, we provide a qualitative evaluation of their key aspects regarding decisions made about architecture and software quality attributes. The framework, which uses the proposed adaptation mechanism, was instantiated in a simple application in the trip domain, in order to test the infrastructure. We plan to perform an experimental study, which consists of using software metrics to compare two system versions: one implemented with our approach (in which the system is adapted) and another that incorporates the logic of satisfying customisations as part of the system, so that we can provide a quantitative evaluation.

We start by discussing, in Section 5.1, the benefits of separating user customisations, which can be seen as requirements, from the implemented system, and using an adaptation process when these customisations change at runtime. Next, in Section 5.2, we provide reasons for the adoption of the BDI architecture for designing and implementing PAS and supporting the adaptation process. Then, we point out in Section 5.3 software quality attributes improved by using

our framework and underlying approach, and present arguments for supporting this claim. Finally, we discuss drawbacks for the adoption of our approach and present its current limitations in Section 5.4.

5.1 Advantages of a Two-Level Architecture

Our previous work has shown that user customisations can be seen as a concern that is spread all over PAS [17]. This is an intrinsic characteristic of preferences because they play different roles in reasoning and action [10]. Systems that adapt their behaviour according to an evolving specification, in our case the user model, must have an architecture that supports variability and its management. This issue is less evident in systems that are concerned only with content customisation, as a single and static architecture is sufficient for providing personalised data, yet the scope of our family of systems is wider than that.

The key advantages of our high-level user model are twofold: it provides a complementary representation that is a global view of user customisations, thus allowing variability management and traceability (captured by rules); and it provides a means for users to understand their model (transparency) and manage it (power of control). Moreover, our two-level abstraction architecture brings additional benefits:

(i) user customisations have an implementation-independent representation;
(ii) the vocabulary used in the user model becomes a common language for users to specify configurations and preferences;
(iii) the user model modularises customisations, allowing modular reasoning about them;
(iv) the user model can be used in mixed-initiative approaches, in which learning techniques can be used to create initial and updated versions of user models, and users have a chance to change them; and
(v) by dynamically adapting PAS, we eliminate unnecessary reasoning (which can be time-consuming) if customisations are represented as control variables that regulate the control flow of the system.

5.2 Benefits of Providing a BDI Agent-Based Design and Implementation

We adopted the BDI architecture to make our adaptation mechanism concrete and used it in our framework. We made this design decision due to the benefits that this architecture can provide to our adaptation process as well as in supporting the implementation of PAS.

However, the most important reason is that the BDI architecture is very flexible. In this architecture, components are loosely coupled and there is an explicit separation between what to do (goals) and how to do it (plans). Goals can be composed to achieve higher-level goals, and plans are independent of each other, and adding or removing them from plan libraries has low impact on the system. A BDI agent has an abstract running cycle, which runs independently of

these fine-grained components and results in a course of actions dynamically composed. The BDI architecture thus facilitates the implementation of user customisations in a modular fashion so that components can be added and removed as the user model changes. A more concrete example is the case of the adaptation rule presented in Section 3.2, in which two plans achieve a goal in two different ways, and the agent behaviour is modified by selecting one of these plans to be part of the agent's plan library, without requiring additional changes to keep the agent functional.

In addition, implementing PAS can be made easier by use of the BDI architecture, which like other agent approaches, is composed of human-inspired components, consequently reducing the gap between the user model (problem space) and the implementation (solution space). Furthermore, plenty of agent-based artificial intelligence techniques have been proposed to reason about user preferences, and can be leveraged to build personalised user agents.

5.3 Software Quality Attributes

By providing a software framework for developing PAS, we also provide a reusable infrastructure to build a family of systems. In addition to following the two-level approach we are proposing, in order to build a high quality architecture, we made design decisions that take into account software quality attributes, as follows.

Reuse. The primary advantage of a framework is reuse, together with its benefits, e.g. higher quality and reliability in a relatively short development time. Using our framework speeds up the process of building PAS systems due to the infrastructure that is ready-to-use and ready to extend, including models that are common in our target application domain. In addition, as we considered good software engineering practices to develop our framework, such as design patterns, this will be inherited by the framework instances.

Maintainability. User customisations are a cross-cutting concern, because they are spread over different points of the application. In our approach, individual customisations are localised in each part of the system that they are related to: if the behaviour of an agent, which is a consequence of a set goals and plans for example, depends on preferences over a particular domain entity, this variability will be encapsulated in that part of the system. At the same time, the high-level user model and rules provide the information needed to trace and manage user customisations as a whole. This modularity of user customisations thus facilitates the maintenance of PAS because software assets of the system have high cohesion and are loosely coupled. For the same reason, this structure reduces the impact of modifying the system, such as adding a new user agent with new services for users.

Scalability. PAS typically involves complex algorithms, which require much processing, such as reasoning about preferences. Running this kind of system with a large number of users at the same time, on a single server, is thus not scalable. As a consequence, we adopted a client-server model to distribute this processing of users across different clients, by still allowing users to

access the application configuration in different clients, making it possible to build different client versions.

Performance. The adaptation mechanism incorporated into our framework adds processing that is executed each time the user model changes. As adaptation rules are application-specific, in cases in which the rules of an application generate a number of actions exponential on the size of changes, we might be adding a significant overhead to the system. However, using adaptation to change the system state avoids reasoning about a set of user customisations *each time* a system behaviour that depends on customisations is executed, and this reasoning process might be costly. A simple example is the use of context for plans in the BDI architecture. A set of user customisations can be related to the context in which plans can be executed. If no adaptation is made, we have two problems. First, a high-level user customisation must be part of the implementation to indicate the context situation. Second, each time a plan must be selected, all the plans will be tested and plans whose contexts do not match with the current situation will be discarded. Using our adaptation mechanism, rules that allow only applicable plans to be part of the plan library of agents will be executed once, i.e. when a change is made in the user model. In addition, the user customisation does not need to appear in the implementation level, because it is only necessary for adding or removing plans of the plan library.

5.4 Drawbacks and Limitations

In the previous section, we showed the advantages, mainly from a software engineering perspective, of our approach. However, both our framework and its underlying approach for PAS have drawbacks and limitations, mainly related to problems already known in the context of SPL and dynamic adaptation.

As mentioned above, a PAS can be seen as a SPL, because it shares a set of commonalities that are provided for all users and variable parts, which are customisations for a particular set of users. SPL engineering involves an initial investment, because it requires (i) scoping what will be part of the SPL, (ii) the analysis of not only a single application, but a family of systems, and (iii) a design that modularises all variable parts of the SPL. This initial investment is usually only amortised by the derivation of the third SPL product [19]. As in our case, we do not derive specific products, but evolve a system at runtime, an alternative to which is the use of planning to satisfy dynamic user customisations. However, for customisations that can be analysed and realised in a relatively easy way, it might be too expensive, as it is not trivial to define the set of necessary actions to compose plans to be formed at runtime, and planning is a hard problem [21].

Another problem that we have not tackled and that is intrinsic to dynamic adaptation is how to make the adaptation *safe* with respect to system consistency [22]. When a system is executing a task, the adaptation has to be made in such a way that the running task is not corrupted, and also that, whenever an adaptation is performed, the system must reach a consistent state. Therefore, a limitation of our approach is that we have not undertaken a consistency analysis

of the adaptation process. The correctness of the adaptation process is related to the correct definition of rules and actions. It is the responsibility of developers to ensure that these are specified in the right way, and therefore the safety of the adaptation process depends on the application-specific rules. In addition, we also do not deal with preference inconsistency, which might arise since users have different forms of expressing preferences.

In addition, our adaptation process generates a set of actions that are performed at the implementation level of PAS. However, we do not consider *order* in such actions. This is important mainly when we have dependencies among features. Until now, our studies have not required consideration of action order, but investigating scenarios in which order matters is part of our future work.

There are also aspects of PAS that are not covered by our approach: learning; security and privacy; and user explanations. Our goal is to extend our framework architecture to accommodate such modules, using this as a reference architecture for PAS. A complete approach for the first two aspects is out of the scope of our research, but we have already taken steps to integrate user explanations into our framework. Even though users can control their user models, there are decisions that agents make on their behalf. Explaining to users the rationale behind decisions is another important factor to increase user trust in PAS.

6 Related Work

In this section we present work that is related to the approach presented in this paper. This is divided into four main areas: (i) agent-based approaches for PAS, (ii) model-based dynamic adaptation; (iii) dynamic adaptation exploited in the context of agents; and (iv) platforms to support dynamic adaptation.

Much research has been carried out in the context of PAS. For example, a multi-agent infrastructure for developing personalised web-based systems, Seta2000 [2], provides a reusable recommendation engine that can be customised to different application domains. Huang et al. [13] describe an agent-based recommender system, providing an implicit user preference learning approach, and distributing responsibilities of the recommendation process among different agents, such as learning agent, selection & recommendation agent and information collection agent. The Cognitive Assistant that Learns and Organizes (CALO) project [3] has also explored different aspects to support a user in dealing with the problems of information and task overload. However, in such work, personalisation in the system is in the *form of data*, so that architecture adaptations are not investigated, which is the main issue addressed in this paper. Indeed, none of this work addresses an evolving system, and consequently systems are not tailored to users' needs in the sense of features that the system provides. In particular, Seta2000 and the work from Huang et al. provide a reusable infrastructure for building web-based recommender systems, but they do not provide new solutions in the context of personalised systems: they leverage existing recommendation techniques and provide implemented agent-based solutions.

A recent trend in the context of model-driven development is to provide runtime adaptation mechanisms that use software models and extend the applicability of model-driven engineering techniques to the runtime environment [4], often referred to as *models@run.time*. As runtime adaptations tend to be complex if they are managed only at the implementation level, models that abstract only the essential information that is needed for the adaptation process can be created and used to drive the adaptation process. This is the main idea of our work, but it differs from existing approaches in being less abstract by adopting a particular model to drive adaptations (end-user model capturing user customisations) and a mechanism to adapt a specific kind of architecture (BDI). There are other existing approaches that use other models, such as that proposed by Floch et al. [11], which follows an architecture-centric approach, in which architecture models are used at runtime to allow generic middleware components to reason about and control adaptation.

Runtime adaptation for agent-based systems has been explored for ubiquitous and mobile environments. Gunasekera et al. [12] propose an approach for adapting multi-agent systems for ubiquitous environments. The paper focuses on using compositional adaptation of individual and teamed software agents to build adaptive systems, and while the adaptation occurs in how agent teams are structured to accomplish a task, there are no fine-grained adaptations, which is an issue relevant in our context. Brandt and Reiser [6] investigate this issue, by providing a methodology for creating mobile agents, able to adapt themselves to the environments in which they are currently running. The main idea is that there are equivalent pieces of software tailored to specific devices, and when an agent migrates from one location to another, the code is loaded according to the new location. The kind of adaptation that is investigated related to only different versions of the software, and does not change the structure of the system, nor the complexity of figuring out which components should be changed.

Recently, Dam and Winikoff [9] have proposed an agent-oriented approach to change propagation in order to maintain software systems. Their approach investigates dependencies in design models and generates a plan for evolving a system based on a requirement change. Making a change in existing software typically requires making other changes to keep the system functional, and this activity is error-prone and time-consuming. Even though this approach also adopts the BDI architecture, the approach *uses* it to solve a software maintenance problem and delivers an analysis only at the design level (horizontal dependency). On the other hand, our approach provides a means for adapting a system based on the BDI architecture and where dependency is from the end-user level to the implementation level (vertical dependency).

Finally, there are some technologies that provide dynamic deployment and adaptation, such as J2EE[4] and OSGi.[5] These technologies are complementary to our approach and to all the approaches above for runtime adaptation as they provide the infrastructure necessary to start new components at runtime, even

[4] http://java.sun.com/j2ee/
[5] http://www.osgi.org/

when there is the addition of new or updated code. However, when and how to adapt is a process independent of these technologies, and helping to manage the complexity of this process is exactly the issue addressed by research work on runtime adaptation.

7 Conclusions

In this paper we have presented a domain-specific model to build high-level user models and a dynamic adaptation mechanism, which together provide a basis for a software framework that is a reusable infrastructure for developing Personal Assistance Software (PAS). The adaptation mechanism allows software customisation not only in term of data, but also in terms of functionality, by evolving a running system based on changes to a user model that drives adaptations. The mechanism was built in a generic way, and extended to our target architecture, the BDI architecture, which provides a flexible structure for development of cognitive agents. Our framework incorporates the idea of providing a two-level view of user customisations, which consists of an end-user high-level model and the realisation of customisations at the implementation level, both associated with the dynamic adaptation mechanism that is responsible for keeping these two levels consistent. Our ultimate goal is to give power of control over task automation to users, and therefore the end-user view is necessary for allowing users to understand how the system is customised and to change it. In addition, in order to provide a large-scale reusable infrastructure to significantly reduce the effort of building PAS, the framework includes a PAS Domain-Specific Model, graphical interface components to manipulate models, support components that provide core functionalities, model persistence, a BDI layer over JADE, and patterns for implementing agents. We have evaluated our approach with a qualitative analysis, identifying its main benefits and software quality attributes.

Our short term future work includes addressing some current limitations of our approach, to deal with the order of actions and user explanations. In addition we recently performed a user study in which we collected about 200 preference specifications that will be used to refine our preference model.

Acknowledgments. This work is partially supported by CNPq 557.128/2009-9 and FAPERJ E-26/170028/2008. It is related to the following topics: Software technologies for web applications - A Multi-Agent Systems Approach for Developing Autonomic Web Applications - G1. Design techniques to improve the development of autonomic Web applications, and Model-driven Design and Implementation of Web Applications - G3. Develop methodologies, empirical studies and tools to support the development of software product lines for the Web context. Simone Barbosa #313031/2009-6 and Carlos Lucena #304810/2009-6 also thank CNPq for respective research grants, and Ingrid Nunes #141278/2009-9 and #201073/2010-2 for financial support.

References

1. Alur, D., Malks, D., Crupi, J.: Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall PTR, Upper Saddle River (2001)
2. Ardissono, L., Goy, A., Petrone, G., Segnan, M.: A multi-agent infrastructure for developing personalized web-based systems. ACM Trans. Internet Technol. 5(1), 47–69 (2005)
3. Berry, P.M., Donneau-Golencer, T., Duong, K., Gervasio, M., Peintner, B., Yorke-Smith, N.: Evaluating user-adaptive systems: Lessons from experiences with a personalized meeting scheduling assistant. In: IAAI 2009, pp. 40–46 (2009)
4. Blair, G., Bencomo, N., France, R.: Models@run.time. Computer 42(10), 22–27 (2009)
5. Boutilier, C., Brafman, R.I., Domshlak, C., Hoos, H.H., Poole, D.: Cp-nets: a tool for representing and reasoning with conditional ceteris paribus preference statements. J. Artif. Int. Res. 21(1), 135–191 (2004)
6. Brandt, R., Reiser, H.: Dynamic Adaptation of Mobile Agents in Heterogenous Environments. In: Picco, G.P. (ed.) MA 2001. LNCS, vol. 2240, pp. 70–87. Springer, Heidelberg (2001)
7. Busetta, P., Howden, N., Rönnquist, R., Hodgson, A.: Structuring BDI Agents in Functional Clusters. In: Jennings, N.R. (ed.) ATAL 1999. LNCS, vol. 1757, pp. 277–289. Springer, Heidelberg (2000)
8. Czarnecki, K., Eisenecker, U.W.: Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., USA (2000)
9. Dam, H., Winikoff, M.: An agent-oriented approach to change propagation in software maintenance. Autonomous Agents and Multi-Agent Systems 23, 384–452 (2011) 10.1007/s10458-010-9163-0
10. Doyle, J.: Prospects for preferences. Computational Intelligence 20, 111–136 (2004)
11. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using architecture models for runtime adaptability. IEEE Software 23(2), 62–70 (2006)
12. Gunasekera, K., Loke, S.W., Zaslavsky, A., Krishnaswamy, S.: Runtime adaptation of multiagent systems for ubiquitous environments. In: Web Intelligence and Intelligent Agent Technologies (WI-IAT 2009), vol. 2, pp. 486–490 (September 2009)
13. Huang, L., Dai, L., Wei, Y., Huang, M.: A personalized recommendation system based on multi-agent. In: WGEC 2008, pp. 223–226. IEEE (2008)
14. Kang, K., Cohen, S., Hess, J., Novak, W.: Peterson: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-021, SEI (1990)
15. Keeney, R.L.: Value-focused thinking – A Path to Creative Decisionmaking. Havard University Press (1944)
16. Malinowski, U., Kühme, T., Dieterich, H., Schneider-Hufschmidt, M.: A taxonomy of adaptive user interfaces. In: HCI 1992, USA, pp. 391–414 (1993)
17. Nunes, I., Barbosa, S., Lucena, C.: An end-user domain-specific model to drive dynamic user agents adaptations. In: SEKE 2010, USA, pp. 509–514 (2010)
18. Nunes, I., Lucena, C., Luck, M.: BDI4JADE: a BDI layer on top of JADE. In: Int. Workshop on Programming Multi-Agent Systems (ProMAS 2011), Taiwan (2011)
19. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005)
20. Rao, A., Georgeff, M.: BDI-agents: from theory to practice. In: ICMAS 1995 (1995)
21. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice-Hall, Englewood Cliffs (2010)
22. Zhang, J., Cheng, B., Yang, Z., McKinley, P.: Enabling Safe Dynamic Component-Based Software Adaptation. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems III. LNCS, vol. 3549, pp. 194–211. Springer, Heidelberg (2005)