Dynamically Adapting BDI Agent Architectures based on High-level User Specifications

Ingrid Nunes[1,2], Simone Barbosa[1], Michael Luck[2], and Carlos 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@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 a software framework that supports the development of Personal Assistance Software (PAS). It relies on the idea of exposing a high-level user model in order to increase user trust in the task delegation process as well as empowering them to manage it. The framework provides a synchronization mechanism that is responsible for dynamically adapting an underlying BDI agent-based running implementation in order to keep this high-level view of user customizations consistent with it.

Keywords: Personal Assistance Software, Framework, User Agents, User Modeling, Adaptation, Software Architecture.

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, e.g. task managers and trip planners, is increasing as the web grows, because people are increasingly facing the challenge of dealing with huge amounts of information and being constantly accessible through mobile devices. In this context, the development of PAS is strongly related to personalization and recommender systems but brings several challenges. In particular, individual user characteristics must be captured to provide personalized content and recommendations for users; i.e. there is a need to elicit, represent and reason about user preferences. In addition, as the typical scenario used in research on preferences is online stores, two key concerns are that users are generally unwilling to provide information and they cannot be expected simply to blindly trust recommendations. Research on implicit learning [4] (learning without users providing explicit information) and explanation interfaces [12] has been addressing these issues, respectively.

Most current research is concentrated on identifying user preferences with elicitation and learning processes, in order to personalize systems solely in terms of data. In our work, we examine a different scenario: we aim to support the development of PAS able to automate routine tasks, in a way that enables configuration directly by users, by choosing from the *services* it provides and customizing them. Such systems must

thus be able to have their architecture adapted dynamically, which is the main issue addressed in this paper. These adaptations must also take into account preferences in order for agents to be able to act appropriately on behalf of users. Furthermore, the ability to understand what the system knows about users and providing them with a means for controlling the system are key issues in automation. Since users directly benefit from interacting with their personal application, they can tolerate spending more time in providing it with information, in comparison to web applications, in which implicit learning is essential, since these are not personal user systems.

In response, in this paper, we introduce a software framework that provides a reusable infrastructure for the development of PAS. The main characteristic of the framework is the adoption of a high-level user model exposed to users (*transparency*), and which they can manage (*power of control*). Here, user customizations are realized by lower level software components, thus we propose a synchronization mechanism that is responsible for dynamically adapting an underlying BDI agent-based running implementation and keeping this high-level view of user customizations consistent with it.

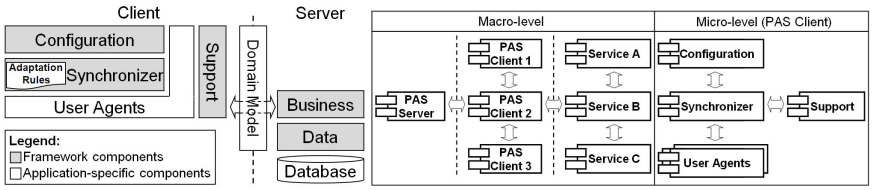
The remainder of this paper is organized as follows. Section 2 describes our PAS framework and our dynamic adaptation mechanism. Section 3 makes a qualitative analysis of our proposal by discussing relevant aspects from it. Section 4 presents related work followed by Section 5, which concludes this paper.

2 A Two-level Framework for Developing PAS

In this section we provide an overview of our framework and detail its two main components: the PAS Domain-specific Model (DSM), which models user customizations in a high-level way, and the synchronization mechanism for dynamic adaptation in belief-desire-intention (BDI) architectures. This architecture provides abstractions and a reasoning mechanism, which are adequate and widely adopted to develop cognitive agents, in particular agents able to automate user tasks. Moreover, this architecture is composed of loosely coupled components and makes an explicit separation between what to do (goals) and how to do it (plans). These two characteristics of the architecture make it very flexible, which facilitates the adaptation process by evolving and changing components with a lower impact in the running system.

2.1 Framework Overview

Our framework is a reusable software infrastructure for developing a family of systems whose goal is to assist users in their routine tasks in a customized way. The main characteristic of our approach is the adoption of two levels of abstractions that capture user customizations: the (end-)user and implementation levels. The first makes user customizations explicit and modularized, as well as understandable by users, so that the current state of the PAS is transparent to users, and empowers them to manage customizations. As users evolve and personalize their PAS over time, and there is an underlying implementation that must be consistent with the user high-level specifications, there is a need to keep both levels synchronized. So changes at the user level drive dynamic adaptations in the underlying implementation, in order for the latter to



(a) The Client-server Architecture. (b) Macro and Micro-level Views.

Fig. 1: PAS architecture.

reach a state consistent with the user-level representation. User *customizations* at the user level are represented in a high-level language, and can be: (i) *configurations*: direct and determinant interventions that users perform in a system, such as adding or removing services and enabling optional features. These configurations can be related to environment restrictions (e.g. a device configuration, or functionalities provided by the system); or (ii) *preferences*: information about user values that influences their decision making, and thus can be used as resources in agent reasoning processes. Preferences typically indicate how a user rates certain options better than others in certain contexts.

The implementation of the 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 customized applications for diverse users. Therefore, all optional parts of the PAS must be modularized in the code, so they can be added and removed from the running application instance. Each set of assets that realizes a variable part of the PAS is the implementation-level representation of user customizations.

Our framework supports the implementation of PAS using a client-server model. User data is stored in a centralized database in the server. The server is structured 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: (i) a *Configuration* layer, which enables users to manage the model that represents customizations at a high-level; (ii) a *User Agents* layer, which implements application-specific services and functionalities for users; (iii) a *Synchronizer* layer, which executes adaptation rules to keep (i) and (ii) consistent; and (iv) a *Support* layer, which provides core PAS services, such as login. The client-server model is illustrated in Figure 1a, highlighting the layers in which the framework is structured. The *Domain Model* is a layer common to both the client and the server sides of the PAS, and is thus shown between them in Figure 1a.

From a macro-level viewpoint, PAS clients can be seen as autonomous and proactive agents that represent users in a multi-agent system. PAS clients communicate to the PAS server to access stored information and other business services, and can communicate with each other. They also communicate with services available on the web. For instance, 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. Figure 1b shows both macro and micro-level views of an instance of our framework.

2.2 PAS Domain-specific Model

The PAS DSM has a key role in our approach. It is a meta-model that defines abstractions to model domain-specific concepts of PAS, such as features and preferences. The goal is to use abstractions close to the vocabulary of users. The PAS DSM was previously defined in [10], however we introduce the parts that are relevant for this paper (see [10] for further details), and highlight improvements over the previous version. The PAS DSM consists of two parts: definition models and the user model. Definition models define abstractions of a PAS that characterize a particular PAS application, e.g. domain entities and provided features. They also serve as a basis for instantiating user models. Abstractions of definition models provide both domain entities (e.g. features and ontology concepts) to be referred to in users models, and restrictions used for defining valid user model instances. In addition, a user model can evolve over time. Definition and user models are instantiated in a stepwise fashion. The first is built by developers during the instantiation of a PAS application, i.e. it consists of *design decisions*. The latter is instantiated at runtime by users (possibly on learning from users), so it corresponds to *user decisions*.

Definition Models. There are three definition models, which are detailed as follows.

Ontology Model. The ontology model defines the set of concepts within the application domain and the relationships between those concepts. It is commonly used for knowledge representation, so its detailed description is out of the scope of this paper. We mention elements of the ontology model while describing our models, which we refer to as: (a) *Classes*: concepts of the ontology; (b) *Properties*: slots of concepts. They can store primitive values or references to other concept instances; (c) *Enumeration*: a set of named values (*EnumerationValue*); and (d) *ValueDomain*: a particular kind of Enumeration, being composed of *Values*. Value [8] is a first-class abstraction that we use to model high-level user preferences. It describes preferences not over characteristics of the object but the value it brings.

Feature Model. This model defines the set of features available for users to configure their PAS application. A feature is any characteristic relevant to the user that, depending on the user configuration, can be part of the application. For instance, it might be a functionality or a setting (e.g. the interface language). Our feature model is an extended and adapted version of feature models used in Software Product Lines (SPLs) [5], which is a new software reuse approach that aims at systematically deriving families of applications based on a reusable infrastructure with the intention of achieving reduced costs and time-to-market. A PAS application can be seen as an SPL whose products are applications customized for a particular user. However, a main characteristic of PAS is providing users with functionalities that automate their tasks, and feature models do not explicit capture it. Therefore, we have enriched feature models by distinguishing a particular kind of feature: the *autonomous features*, which provide a functionality of acting on behalf of users for performing a user task.

Each autonomous feature has a set of autonomy degrees associated with it, which means the different degrees of autonomy that the feature makes available to the user.

Possible autonomy degrees are `initiate`, `suggest`, `decide` and `execute`, which were defined based on the taxonomy presented in [9]. The availability of these autonomy degrees does not imply that this automation will be performed; this is determined in the configuration of the user model. An example is a *Flight Planner* feature that may be able to `suggest` and `execute` the process of buying a flight ticket for a user, but not to `initiate` and `decide` about it.

Preferences Definition Model (PDM). Because it is desirable that users are able to express preferences in different ways, it is necessary to have a system that can deal with them. For instance, if an application can deal only with quantitative preferences, preferences expressed in a qualitative way will have no effect on the system behavior if there is no mechanism to translate them to quantitative statements. The PDM specifies restrictions over preferences users can express, i.e. the purpose of this model is to specify *how* users can express preferences and *about which elements* of the ontology model.

Users can provide different kinds of preferences statements in the user model, which were chosen using preference statements collected from different individuals and from existing models to reason about preferences. The aim was to consider the different kinds of preference statements in order to maximize the expressiveness of users. There are five different kinds of statements: *order* (an order relation between two elements), *rating* (users attribute a rate to a target), *minimization* or *maximization* (user preference is to minimize or maximize an element), *reference value* (users indicate one or more preferred values for an element) and *don't care* (a set of elements the user does not care about, they are equally (un)important to the user). A preference target can be associated with a subset of these kinds of preference statements, so that it is possible to express those kinds of preference about the target. There is an exception for rating preferences, whose definition is based on rating domains: rating preferences related to a target can be expressed if that target is associated with a rating domain, and the rating must belong to the defined domain. This domain can be numeric (either continuous or discrete), with specified upper and lower bounds; or an enumeration. Targets can be one of four types, which are those described in the ontology model.

User Model. The user model specifies user customizations for each individual user, and is built on abstractions from the feature model and the PDM of an application. These models are used to constrain the user model instance. As stated before, user customizations can be either configurations or preferences, so the user model is composed of two parts: a configuration and a set of preferences. As the user model is managed by users, the *Configuration* layer of the framework provides a graphical interface to manipulate it. The configuration comprises a set of features that are selected from the feature model, and a set of feature autonomy configurations. The set of selected features must be valid according to the feature model. A feature autonomy configuration stores the autonomy *degree* that a user defines for an autonomous feature, i.e. a feature whose autonomy degree set is not empty. Therefore, the feature autonomy configuration associated with each autonomous feature is a subset of autonomy degrees that it provides.

2.3 Dynamically Adapting PAS Clients

Our approach requires synchronization between the user and implementation levels, because the former represents user customizations at a high-level and the latter must re-

flect these customizations. This is achieved by an adaptation process, which is triggered by changes performed in the user model. Figure 2a 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, and the ontology model provides users with a vocabulary to make preference statements. Based on these two models, users create or *evolve* a user model, and it is then validated according to the feature model, which also contains constraints over selected features, and the PDM. A preference statement is valid if the target has the associated allowed preference (or a rating domain, in case of rating preferences) defined in the PDM, or if there is not definition in the PDM, since the default is that all preference types are allowed.

When the user model changes, the synchronization mechanism causes the implementation-level of the PAS client to be consistent with it. The mechanism consists of knowledge that captures how the implementation level must be adapted according to changes in the user model and a process that uses this knowledge to adapt the implementation level as the user model evolves. This knowledge is composed of four main concepts:

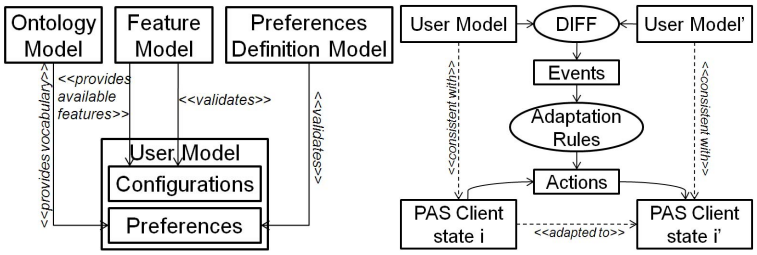
Events correspond to changes that occur in the user model, for instance adding or removing the autonomy degree of a feature.

Event Categories group a set of related events. For example, the events above belong to the autonomy degree category.

Actions are the changes to be performed over software components at the implementation level, e.g. adding or removing agents, beliefs or goals. Here, a *software component* is any software asset that is part of the implemented system, and there are two types of coarse-grained software components: *components* and *agents*. The former provides a reactive behavior, while the latter provides autonomy and pro-activity, has its own thread of execution, and is able to communicate through messages with other agents. As a result, agents are composed of finer-grained software components, namely capabilities, beliefs, goals and plans, required parts of the BDI architecture [13], which we adopt to design and implement agents – this design decision is discussed later.

Rules establish connections between events and actions, and are applied when an event in 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, 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. For instance, let $R1$ be a rule stating that agent $A1$ must be part of the running system if features $F1$ and $F2$ are selected. Then, if the event $Feature(F1, Add)$ occurs, $R1$ generates the action $Agent(A1, Add)$ only if feature $F2$ was previously selected or the event $Feature(F2, Add)$ also occurres.

These concepts are used in our adaptation process, shown in Figure 2b, implemented as part of the `Synchronizer` agent (Figure 1b). The process uses as input a previous and an updated version of a user model, and a set of adaptation rules. A new user model (initial state) is a configuration with the core features selected and no preference statements. 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



(a) Relationships among models. (b) Adaptation Process.

Fig. 2: Evolving a PAS architecture.

```
interface Event
    Set<EventCategory> getEventCategories()

interface EventCategory

interface AdaptationAction
    void doAction() throws AdaptationException

interface AdaptationRule
    Set<EventCategory> getObservableEventCategories()
    List<AdaptationAction> getAdaptationActions(
        UserModel oldUserModel, UserModel newUserModel,
        Set<Event> events) throws AdaptationException
```

(a) Interfaces of our Mechanism.

```
rule AutonomyDegree (
    eventCategory AutonomyDegree (
        feature = AccommodationPlanner,
        autonomyDegree = DECIDE),
    ONA = {
        action Plan (
            capability = AccommodationService,
            plan = DecideAccommodationPlan) },
    OFFA = {
        action Plan (
            capability = AccommodationService,
            plan = DecideAccommodationUserPlan),
        action Belief (
            capability = AccommodationService,
            plan = DecideAccommodationActionBelief) })
```

(b) Autonomy Degree Adaptation Rule.

Fig. 3: Framework interfaces and adaptation rule.

of each set of actions generated by each selected rule. Finally, each action is executed. The complexity and performance of the algorithm that implements this process is not discussed because it depends of how application-specific rules generate actions.

It can be seen that the process is simple in that it does not specify any concrete event, event category, rule or action. It is developed in this way to address any instance of any one of these concepts. They are built in the framework as interfaces (Figure 3a), therefore one can create concrete classes by implementing such interfaces. Therefore, we have a generic structure to make adaptations. The goal of providing a generic structure is to make it extensible. However, we also provide a set of predefined events, event categories, rules and actions as part of our framework, presented in Table 1. Actions allow manipulation on software components of the BDI architecture. Due to space restrictions we do not describe each of the elements of this set, but give an overview by explaining one of the rules – the $FeatureExpression$ rule. This rule receives as parameter a logic formula expression, in which literals are features. The rule is associated with a set of event categories composed of $Feature(F)$ for each literal F of the formula. Then, the generated set of actions will be the following: if the formula is evaluated to a $false$ value in the previous version of the user model and a $true$ value in its updated version, the set of actions will be the *ONA* set with action operators set to add together with the *OFFA* set with the action operators set to $remove$. If the formula was evaluated to

Events	Rules
AutonomyDegree(F, AD, ET) Feature(F, ET) Preference(P, ET)	AutonomyDegree(ADC, ONA, OFFA) FeatureExpression(LF, ONA, OFFA) OptionalFeature(F, ONA, OFFA) *Preference(PEC)* – Abstract
Event Categories	**Actions**
AutonomyDegree(F, AD) ClassPreference(Class) EntityPreference(Class) EnumPreference(Enum) EnumValuePreference(EnumValue) Feature(F) *Preference* – Abstract PropertyPreference(Property) ValueDomainPreference(VD) ValuePreference(V)	Agent(A, AO) Belief(C, B, AO) BeliefSetValue(C, B, O, AO) BeliefValue(C,B,O) Capability(A, C, AO) Component(Female, Male, AO) ComponentValue(Female, Male) Goal(A, G, AO) Plan(C, P, AO)
Legend: ET: Event Type (Add, Remove); F: Feature; AD: Autonomy Degree, P: Preference; ADC: Autonomy Degree Category; VD: Value Domain; V: Value; ONA: On Action Set; OFFA: Off Action Set; LF: Logic Formula; PEC: Preference Event Category; AO: ActionOperator (Add, Remove, Set); A: Agent; C: Capability; G: Goal, B: Belief; P: Plan; O: Object.	

Table 1: Predefined Set of Events, Event Categories, Rules and Actions.

$true$ and $false$ respectively, the action operators would be inverted. Figure 3b shows the information that is defined in a rule, which is applied when the autonomy degree `DECIDE` of the `AccommodationPlannerFeature` changes.

2.4 Implementation Details

Several agent platforms implement the BDI architecture, e.g. Jason, Jadex, 3APL and Jack. Most of these are based on Java, as in our framework, but 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, that can help with the implementation of our adaptation mechanism, and complicate integration with other frameworks. Due to this limitation of existing BDI agent platforms, we have developed BDI4JADE [11], a BDI layer on top of JADE[3] (Java-based agent platform that provides a robust infrastructure to implement agents, including behavior scheduling, communication and yellow pages service). We have leveraged these features, provided the BDI abstractions, and built a BDI reasoning mechanism for JADE agents. Agents developed with our JADE extension are implemented only in Java, and not by using additional files in XML, for example, as used in Jadex. As BDI4JADE components are extension 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.

This was needed for supporting the implementation of our adaption mechanism, which is extensively based on the use of the Spring framework,[4] a Java platform that provides comprehensive infrastructure support for developing Java applications. It is designed to be non-intrusive, so the domain logic code generally has no dependencies on the framework itself. Mostly, we took advantage of the Dependency Injection

[3] http://jade.tilab.com/

[4] http://www.springsource.org/

and Inversion of Control module, which allows declaration of the application software components (*beans*, in the Spring terminology) and dependencies among them. Thus, actions in our synchronization mechanism receive strings (bean identifiers) as parameters, referring to software components of the running PAS to be adapted. These bean declarations can correspond to either a singleton or a prototype instance of the bean. In addition, rules and actions are also declared in the Spring configuration file.

3 Discussion: a Qualitative Analysis of our Approach

As an initial step for the evaluation of our framework, we provide a qualitative evaluation of key aspects of our framework regarding decisions made about architecture and software quality attributes. The framework was instantiated in a simple application in the trip domain, in order to test the infrastructure. As our long term research aims to increase user trust in PAS by exposing user models at a high level, part of our future work will involve a user study to validate this aspect of our approach.

3.1 Advantages of a Two-level Architecture

Our previous work has shown that user preferences and configurations can be seen as a concern that is spread all over PAS [10]. This is an intrinsic characteristic of preferences because they play different roles in reasoning and action [6]. Systems that adapt their behavior 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 customization, as a single and static architecture is sufficient for providing personalized data, yet the scope of our family of systems is wider than that.

The key advantages of our high-level user model are twofold: (i) it provides a complementary representation that is a global view of user customizations, thus allowing variability management and traceability (captured by rules); and (ii) 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 customizations 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 modularizes customizations, 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 customizations are represented as control variables that regulate the control flow of the system.

3.2 Benefits of Providing a BDI Agent-based Implementation

Our framework uses software agents at the implementation level of PAS, following the BDI architecture. We made this design decision due to the benefits of adopting an agent-based approach. 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). Thus, evolving and changing an agent architecture is easier in comparison with objects. The BDI architecture thus facilitates the implementation of user customizations in a modularized way so that components can be added and removed as the user model changes. The adaptation rule presented in Figure 3b helps to illustrate this situation: for making an accommodation reservation (top level goal), the agent has to achieve three subgoals – search and suggest, decide and execute. There are two ways of deciding for an accommodation: (i) asking the user to decide; or (ii) deciding on behalf of the user. Simply by changing the plan that is part of the agent plan library, the agent behavior is changed without impacting the course of actions or other actions to be executed.

In addition, the BDI architecture and other agent approaches are composed of human-inspired components, thus reducing the gap between the user model (problem space) and the solution space. Furthermore, plenty of agent-based artificial intelligence techniques have been proposed to reason about user preferences, and can be leveraged to build personalized user agents.

3.3 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 customizations are a cross-cutting concern, because they are spread over different points of the application. In our approach, individual customizations are localized in each part of the system that they are related to: if a behavior of an agent (plans, goals) depends on preferences about *X*, 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 customizations as a whole. This modularization of user customizations thus facilitates the maintenance of PAS because software components of the system have high cohesion and are loosely coupled. For the same reason, this structure reduces the impact of evolving 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, thus is 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.

3.4 Limitations

Both our framework and its underlying approach for developing PAS have some limitations. Our synchronization algorithm generates a set of actions that are performed at the implementation level of PAS. Nevertheless, we do not consider *order* in such actions. This is important mainly when we have dependencies among features. Up to now, our studies have not required consideration of action order, but investigating scenarios in which order matters is part of our future work.

In addition, we have not considered the consistency analysis of the adaptation process and user preferences. The correctness of the adaption process is related to the correct definition of rules and actions. It is responsibility of developers to ensure that they are specified in the right way. In addition, users have different forms of expressing preferences, which might be inconsistent. We do not provide mechanisms for detecting such situations.

There are 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.

4 Related Work

Much research has been carried out in the context of PAS. For example, a multi-agent infrastructure for developing personalized web-based systems, Seta2000 [2], provides a reusable recommendation engine that can be customized to different application domains. Huang et al. [7] describe a personalized recommendation system based on multiple-agents, providing an implicit user preference learning approach, and distributing responsibilities of the recommendation process among different agents, such as learning, selection & recommendation 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, personalization 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. None of them address an evolving system, and consequently systems are not tailored to users' needs in the sense of features that the system provides. In particular, [2] and [7] provide a reusable infrastructure for building web-based recommender systems, but they do not provide new solutions in the context of personalized systems: they leverage existing recommendation techniques and provide implemented agent-based solutions.

5 Conclusion

In this paper we have presented a software framework and a dynamic adaptation mechanism, which provide a reusable infrastructure for developing Personal Assistance Software (PAS). The main idea underlying our framework is to provide a two-level view of user customizations: an end-user high-level model and the realization of customizations at the implementation level. The end-user view aims to give power of control over task automation to users. The framework aggregates a dynamic adaptation mechanism that is responsible for keeping these two levels consistent, and comprises: a PAS Domain-specific Model, a synchronization mechanism, graphical interface components to manipulate models, support components that provide core functionalities, models 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, which are dealing with the order of actions and user explanations. In addition, recently, we performed a user study in which we collected about 200 preferences specifications that will be used to refine our preferences model.

References

1. Alur, D., Malks, D., Crupi, J.: Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall PTR, Upper Saddle River, NJ, USA (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'09. pp. 40–46 (2009)
4. Claypool, M., Le, P., Wased, M., Brown, D.: Implicit interest indicators. In: Proceedings of the 6th international conference on Intelligent user interfaces. pp. 33–40. IUI '01, ACM, New York, NY, USA (2001)
5. Czarnecki, K., Eisenecker, U.W.: Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., USA (2000)
6. Doyle, J.: Prospects for preferences. Computational Intelligence 20, 111–136 (2004)
7. Huang, L., Dai, L., Wei, Y., Huang, M.: A personalized recommendation system based on multi-agent. In: WGEC '08. pp. 223–226. IEEE (2008)
8. Keeney, R.L.: Value-focused thinking – A Path to Creative Decisionmaking. Havard University Press (1944)
9. Malinowski, U., Kühme, T., Dieterich, H., Schneider-Hufschmidt, M.: A taxonomy of adaptive user interfaces. In: HCI'92. pp. 391–414. USA (1993)
10. Nunes, I., Barbosa, S., Lucena, C.: An end-user domain-specific model to drive dynamic user agents adaptations. In: SEKE'10. pp. 509–514. USA (2010)
11. Nunes, I., Lucena, C., Luck, M.: BDI4JADE: a BDI layer on top of JADE. In: 9th International Workshop on Programming Multi-Agent Systems (ProMAS 2011). Taipei, Taiwan (2011), to appear.
12. Pu, P., Chen, L.: Trust-inspiring explanation interfaces for recommender systems. Know.-Based Syst. 20, 542–556 (August 2007)
13. Rao, A., Georgeff, M.: BDI-agents: from theory to practice. In: ICMAS'95 (1995)