Supporting the Development of Personal Assistance Software

Ingrid Nunes, Simone D.J. Barbosa, Carlos J.P. de Lucena

[1]Pontifical Catholic University of Rio de Janeiro (PUC-Rio) – Rio de Janeiro, RJ – Brazil
{ionunes,simone,lucena}@inf.puc-rio.br

Abstract. *In this paper we present a framework for developing personal assistance software, which follows a reference architecture previously proposed for this application domain. The framework includes the implementation of a user metamodel, which allows modeling of users' preferences and configurations using high-level abstractions, typically used in users' vocabulary, thus abstracting from the underlying implementation model. Moreover, the framework provides a mechanism to adapt applications at runtime based on changes on a user model, in order to keep the consistency between user and implementation models.*

1. Introduction

Recently, Rogoff claimed that the Artificial Intelligence (AI) area is going to be the next big driver of global growth [Rogoff 2010]. He claims that computers are going to automatically perform a growing range of tasks in the next 50 years. Multi-agent Systems (MASs), with roots not only in AI but also in distributed systems and software engineering, have addressed this application domain, including auction staging, mission scheduling and e-commerce. In such applications, it is not uncommon the existence of personal agents representing users in the MAS and acting pro-actively on their behalf. Given that agents represent individuals in these scenarios, there remains a need to personalize the software system to meet specific needs of the users [Nunes et al. 2009].

Extensive research work has been carried out in the context of user agents as personal assistance software. It includes eliciting preferences, learning them by monitoring users and reasoning about preferences. Even though these works have significantly advanced the area of user agents, little research effort has been done regarding the engineering of these systems. Good (modular, stable, ...) architectures are essential to produce software with higher quality and easier to maintain. Otherwise, software architectures may degenerate over time, making their maintenance a hard task, by increasing costs with refactorings. Since the late 1980s, software architecture has emerged as the principled understanding of the large-scale structures of software systems [Shaw and Clements 2006].

In our previous work [Nunes et al. 2010], we have proposed a software architecture to build personal assistance software based on agents. It emerged from the analysis of current software engineering practices adopted to implement this domain of applications as well as with the goal of empowering users to control their agents. In this paper, we take one more step for providing a substantial infrastructure to support the development of personal assistance software. We have developed the first version of a framework based on our architecture. The framework provides: (i) the implementation of the user metamodel detailed in [Nunes et al. 2010], which is ready to be instantiated and persisted; (ii) a mechanism to adapt applications at runtime based on changes on the user model. In addition, due to limitations of existing agent platforms, we provide a belief-desire-intention (BDI) extension of the JADE platform that is used combined with our adaptation mechanism.

The remainder of this paper is organized as follows. Section 2 describes the reference architecture that guided the development of our framework. Section 3 presents and details the framework we have been developing, highlighting its adaptation mechanism. Relevant discussions about our framework are presented in Section 4. Related work is presented in Section 5, followed by Section 6, which concludes this paper.

2. The Reference Architecture

In our previous work [Nunes et al. 2010], we have proposed a software architecture to build user-model-based systems, depicted in Figure 1. This architecture emerged from an analysis of software engineering issues of this kind of system. The architecture relies on the concept of *virtual separation of concerns* [Kästner and Apel 2009]. The main idea is to structure the architecture of user-customizable personal assistance systems in terms of service-oriented agents and physically modularize each variability as much as possible into agent abstractions. In addition, we provide a high-level user model. This model is virtual because it is not physically implemented into code, but provides a modularized view of user customizations. User preferences and configurations are not design or implementation abstractions, but they are implemented by typical agent abstractions (beliefs, goals, plans, etc.), i.e., they play their specific roles in the agent architecture. The virtual user model is a complementary view that provides a global view of user customizations. This model uses a high-level end-user language, through which users are able to configure their agents. Next, we briefly describe each one of it modules.

Domain Model. It is composed of entities shared by user agents and services, application-specific, etc.

User Model. This module contains user configurations and preferences expressed in a high-level language.

User Agents. It consists of agents that provide different services for users, e.g. scheduling and trip planning. Their architecture supports variability related to different users, and provides mechanisms for reasoning about preferences. These agents use services provided by a distributed environment (the *Services* cloud), and their knowledge is based on the *Domain Model*.

Synchronizer. The connection between the *User Model* and *User Agents* is stored in the form of trace links, indicating how and where a customization is implemented in one or more user agent. Adaptations are performed at runtime and are accomplished based on the trace links between the *User Model* and the *User Agents*

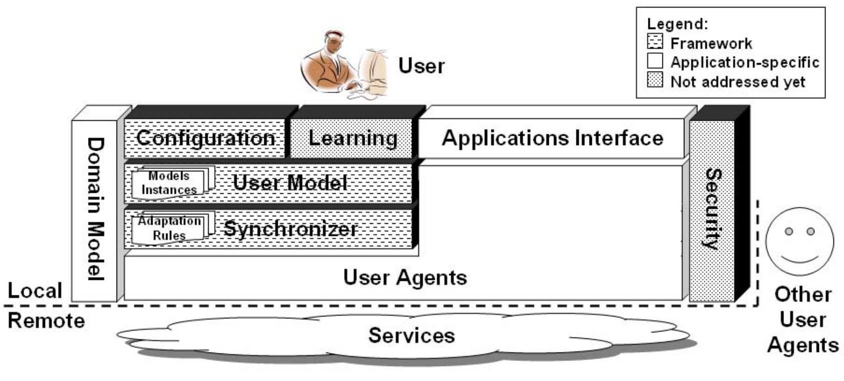

Figure 1. Software architecture to build user-model-based systems.

architecture. The *Synchronizer* is the module in charge of adapting *User Agents* based on changes in the *User Model*.

Applications Interface. Users access user agents' services through this module.

Configuration. By means of the *Configuration* module, users can directly manipulate the *User Model*, which gives them the power to control and dynamically modify user agents, using a high-level language.

Learning. Changes in the *User Model* may be performed or suggested by the *Learning* module, which monitors user actions to infer possible changes in the *User Model*.

Security. It aggregates policies that restrict this communication, assuring that confidential information is kept safety secured.

3. A Framework for Developing Personal Assistance Software

In this section, we present the framework we have been building in order to support the development of personal assistance software. The framework follows the architecture presented in the previous section, however some of its modules (learning and security) have not been addressed yet, as Figure 1 shows. Our framework covers the User Model, Synchronizer and Configuration modules.

In this paper, we focus on detailing the implemented user model (Section 3.1) and on the adaptation mechanism (Section 3.2) that keeps the system implementation consistent with the current state of the user model. In addition, we present an implementation of the BDI architecture [Rao and Georgeff 1995] that we have built on top of JADE[1] framework that is used by our adaptation mechanism (Section 3.3).

3.1. User Model

The User Model plays a key role in our framework. It models how an assistance software is personalized to a user. It is important to highlight that this model uses high-level abstractions, typically found in users' vocabulary, thus abstracting from the underlying implementation model and empowering the user to customize the application.

Our approach distinguishes user *configurations* from *preferences*, which we collectively refer to as *customizations*. Configurations are direct and determinant interventions that users perform in a system, such as adding/removing services or enabling optional features. They can be related to environment restrictions, e.g., a device configuration, or functionalities provided by the system. Preferences represent information about the users' values that influence their 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. System changes of this nature can be exemplified by the addition/removal of plans from a user agent plan library, which must be in accordance with how a user would act for accomplishing a certain goal.

Our framework provides an implementation of the User metamodel, which we have proposed in [Nunes et al. 2010]. This metamodel is ready to be instantiated. This instantiation is accomplished in a stepwise fashion, first by specific-application developers (development time) and then by users (runtime), as detailed in the remainder of this section. Our framework also provides persistence to these models as well as a software component to be added to Java Swing[2] applications to manage user models at runtime.

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

[2]http://java.sun.com/docs/books/tutorial/uiswing/

Preference	Description	Example
`DontCare`	Allows indicating a set of elements the user does not care about.	<target1> ... <targetN> are indifferent for me.
`MaxMin`	Indicates that the user preference is to minimize or maximize a certain element.	I prefer to maximize/minimize <target>.
`Order`	Expresses an order relation between two elements. A set of instances of the Order preference comprises a partial order	I prefer <target1> to <target2>.
`Rating`	Allows users rating an element. By defining a `RatingDomain` for an element, users can rate this element with a value that belongs to the specified domain. This domain can be numeric (either continuous or discrete), with specified upper and lower bounds. In addition, an enumeration can be specified.	I rate <target> with the value <rate>.
`ReferenceValue`	Enables users to indicate one or more preferred values for an element. It can be interpreted as the user preference is a value on the order of the provided value.	I prefer <target> as close as possible to <reference value>.

Table 1. Preference statements represented in our approach.

The User metamodel is composed of four parts: (a) Variability model; (b) Preferences Definition Model; (c) Variability Model Configuration; and (d) Preferences Model. (a) and (b) are instantiated by developers and define rules for the User Model. (c) and (d) comprise the User Model represent users' configurations and preferences, respectively, and must be in accordance with (a) and (b). The User Model can be managed by users or by the learning module. The Variability model allows modeling variable traits within the domain, which are used for defining user configurations. It defines the possible configurations of the system. It consists of variable features, which are the system-specific characteristics that can be chosen by users. These features can be optional or alternative. Alternative features are organized into feature groups, which define the number of alternative features that can be chosen in a group. In addition, constraints can be defined to establish valid combinations of these features.

The purpose of the Preferences Definition model is to define *how* (preferences statements) users can express their preferences and *about which elements* (preference targets) of the Ontology Model, which is part of the Domain model and defines the concepts and their slots used in the system. Even though it is desirable that users be able to express preferences in different ways, it is necessary to have agents that can deal with them. For instance, if application agents can deal only with quantitative preference statements, user preferences expressed in a qualitative way will have no effect on the system behavior.

Our framework provides five types of preference statements (Table 1). The current version of the framework does not support the specification of conditions over the preferences, as it is defined in the metamodel proposed in [Nunes et al. 2010]. As shown in Table 1, preferences statements contains targets, which are the objects referred in the statement. Table 2 describes the four kinds of preferences target supported.

The User Model is instantiated based on our metamodel and these two described models. The Variability Model Configuration consists of a set of chosen optional and alternative features, and the Preferences Model is a set of preference statements. The current version of our framework verifies if the Variability Model Configuration is consistent with the Variability Model, and if the Preferences Model is consistent with the Preferences Definition Model. However, we do not provide a mechanism to detect inconsistencies among preferences, e.g., I prefer A to B, I like B, and I dislike A.

Preference	Description	Example
`Class`	It refers to classes (or concepts) of the Ontology model.	I prefer notebook to desktop.
`Property`	It refers to properties (or slots) of classes. It can refer only to the property (*color of a notebook*) or the value of the property (*notebook color is pink*). In addition, the property can be referred in a nested context, e.g. *Notebook.screen.size*.	I don't like notebooks whose color is pink.
`EnumerationValue`	It refers to values of an enumeration.	I prefer red to blue. (Enumeration is Color)
`Value`	It refers to values of a value domain. Value is a first-class abstraction of our model. It describes preferences not over characteristics of the object but the value it brings.	Cost is more relevant than quality.

Table 2. Preference targets that can be referenced by preference statements.

3.2. Adaptation Mechanism

In our proposed architecture, and consequently in our framework, there are two representations of a system and its current state, which are in different levels of abstractions: (i) User model – which represents users' configurations and preferences in a high-level language and can be managed by the end-user; and (ii) User Agents – an implementation level representation of the system. It consists of software components that implement all user customizations, but are configured according to a specific state of the user model. Therefore, a set of customizations represented in the User Model in a high-level of abstraction is implemented, and represented, with lower level abstractions, such as components, agents, beliefs and plans.

At this point it is important to define some terms used to detail our adaptation mechanism. A *software component* is any software asset that is part of the implemented system. There are two types of coarse-grained software components: *components* and *agents*. The former presents a reactive behavior, as opposed to the latter, which presents autonomy and pro-activity as well as is able to communicate through messages with other agents. Agents are composed of finer-grained software components, namely capabilities, beliefs, goals and plans.

The implementation level representation, i.e., User agents, is mandatory in the system – it is how the system is implemented and shows the expected behavior. On the other hand, the User model is a complementary representation that provides a global view of user customizations and abstracts implementation details, which are not necessary to define specific user customizations. This model brings the following advantages: (i) it provides a common language for users to specify their customizations; (ii) it helps on mixed initiatives in which users can verify and adjust customizations inferred by the system; and (iii) it provides a modular reasoning about user customizations, which is performed using high-level abstractions and is independent of implementation technologies.

However, given that we have two representations of the same system, there is a need for keeping them consistent. The User Model drives the system behavior and determines how the current state of the system should be, and User agents must be configured to be in accordance with it, and be adapted when the User Model changes. Our framework provides an adaptation mechanism, which is responsible for keeping this consistency.

The algorithm that is performed to accomplish this task is presented in Algorithm 1. It receives as input the previous and the updated versions of the User Model as well as a map of adaptation rules. The main idea of the algorithm is to generate and perform a set of adaptation actions that must be performed in the system according to changes in the two versions of the user model.

Algorithm 1: Adaptation algorithm

```
Input: $UM$: previous user model; $UM'$: updated user model; $rulesMap$: adaptation rules mapped to the events
        they observe
$events$ = diff($UM, UM'$);
$adaptationRules = \emptyset$;
foreach Event $e \in events$ do
    $rules$ = $e$.get($rulesMap$);
    if $rules \neq null$ then
        $adaptationRules = adaptationRules \cup rules$;
$actions = \emptyset$;
foreach Rule $r \in adaptationRules$ do
    $actions = actions \cup$ $r$.getAdaptationActions($UM$, $UM'$, $events$);
foreach Action $a \in actions$ do
    $a$.doAction();
```

The core of our adaptation algorithm is the adaptation rules. An adaptation rule is an interface that defines the following methods:

- `Set<Event> getObservableEvents();` – Provides the set of events that this rule is listening to. This method is used to build the map of adaptation rules (a map from events to the rules that listen to them); and
- `List<AdaptationAction> getAdaptationActions(UserModel oldUserModel, UserModel newUserModel, Set<Event> events) throws AdaptationException;` – Generates the list of actions that must be performed according to the changes in the User Model.

An adaptation action, in turn, is an interface that defines the method `void doAction() throws AdaptationException`. In Figure 2, we illustrate an example of a simple rule, the *Optional Feature* adaptation rule, which adds and removes components, agents, etc., based on the addition or removal of a feature. The rule is associated with a feature F and a set of `ComponentAdaptationAction`. It listens to two kinds of event: AddFeature(F) and RemoveFeature(F). The framework provides a set of types of rules and actions, which can be extended for specific applications. However, *instances* of rules and actions are application-specific. The predefined set of rules and actions is still under development. Figure 2 presents two actions: one that adds/removes components and another that adds/removes agents.

It is important to notice that actions are represented with `Strings`. We have made extensive use of the Spring framework[3]. It provides a mechanism for defining software components (`Beans`) and establishing dependencies among them in a XML file. The actions are referenced in the *Optional Feature* adaptation rule by its name in the Spring definition file. In addition, all application-specific rules and actions are defined in this file. They are instances of the rules (e.g. Optional Feature and Preference Statement) and actions (e.g. add/remove component, agent, belief and plan) that our framework provides.

3.3. A BDI layer for JADE

As presented previously, we have adopted an agent-based solution to design and implement the application domain we are addressing, i.e. personal assistance software. We

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

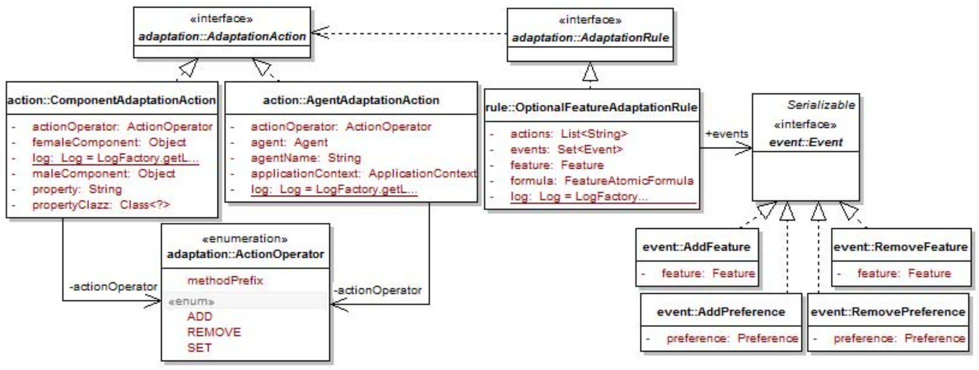
«interface»
adaptation::AdaptationAction
«interface»
adaptation::AdaptationRule
action::ComponentAdaptationAction
- actionOperator: ActionOperator
- femaleComponent: Object
- log: Log = LogFactory.getL...
- maleComponent: Object
- property: String
- propertyClazz: Class<?>
action::AgentAdaptationAction
- actionOperator: ActionOperator
- agent: Agent
- agentName: String
- applicationContext: ApplicationContext
- log: Log = LogFactory.getL...
rule::OptionalFeatureAdaptationRule
- actions: List<String>
- events: Set<Event>
- feature: Feature
- formula: FeatureAtomicFormula
- log: Log = LogFactory...
+events
Serializable
«interface»
event::Event
«enumeration»
adaptation::ActionOperator
methodPrefix
«enum»
ADD
REMOVE
SET
-actionOperator
-actionOperator
event::AddFeature
- feature: Feature
event::RemoveFeature
- feature: Feature
event::AddPreference
- preference: Preference
event::RemovePreference
- preference: Preference

Figure 2. Optional Feature adaptation rule.

have made this choice due to several reasons: (i) agent-based architectures are composed of human-inspired components, such as goals, beliefs and motivations, thus reducing the gap between the user model (problem space) and the solution space; (ii) plenty of agent-based AI techniques have been proposed to reason about user preferences, and they can be leveraged to build personalized user agents; and (iii) agent architectures are very flexible, thus facilitating the implementation of user customizations. For instance, there is an explicit separation of what to do (goals) from how to do it (plans).

We have adopted the BDI architecture [Rao and Georgeff 1995] to design and implement agents. This architecture is a widely used agent architecture and is highly inspired by human reasoning model, thus being in accordance with our argument (i). Several agent platforms that implement the BDI architecture have been proposed, for instance Jason, Jadex and Jack. In particular, these three platforms are based on the Java language. However, agents are implemented in these platforms in a Domain-specific Language (DSL) in specific file types (e.g. XML), which are processed and run in the Java platform. As a consequence, this prevents us from using some features of the Java language, such as reflection and annotations, that help on the implementation of our adaptation mechanism.

Due to this limitation of these BDI agent platforms, we have implemented a BDI layer on top of JADE. JADE is a 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 by JADE, and built a BDI reasoning mechanism to JADE agents. Agents developed with our JADE extension are implemented in "pure" Java language, i.e. not in XML files. These are some of the main characteristics of our JADE extension:

- *Use of capabilities.* Agents aggregate a set of capabilities, which are a collection of beliefs and plans. This allows modularizing particular agents' behaviors.
- *Java generics for beliefs.* Beliefs can store any kind of information and are associated with a name. If the value of a belief is retrieved, it must be cast to its specific type. We have used Java generics to capture incorrect castings in compile time.
- *Plan bodies are instance of JADE behaviors.* In order to better exploit JADE features, in particular its behaviors hierarchy, plan bodies in our extension are subclasses of JADE behaviors. To make this possible, they have to implement the `PlanBody` interface of our extension as well.

Due to space restrictions, we do not present the class diagram that represents our JADE extension. We refer the reader to [Nunes 2010] for further details of it, including its download.

4. Discussions

In this section, we present relevant discussion related to the reference architecture we are implementing and the framework we have been developing.

Relationship with Software Product Lines. Software Product Lines (SPLs) [Pohl et al. 2005] are becoming an essential software reuse technique that has been applied in the construction of mass-produced software systems. A SPL is a family of related software products built from a common set of software components, where each component typically implements a distinguishable feature. SPLs exploit common and variable features of a set of systems and lead to the development of flexible architectures that support the derivation of customized systems. Customized user agents can be seen as a SPL of user agents that are tailored to specific users. As a consequence, SPL techniques are used to support the development of this family of agents. However, the main difference between personalized user agents and SPLs is that the latter contemplates only user configurations, but not user preferences, which are essential to make a system that is able to act on the users' behalf. In addition, our adaptation rules and actions do not require a full mapping between variation points of the Variability Model to the elements that implement them. There is only the need for establishing the actions that must be performed in order to (dis)connect software components. No action needs to be taken regarding software components that are associated with the one being (dis)connected and are not bound to the rest of the running system.

In our framework, all software components are loaded in the application startup, and their connections evolve over time. However, it is interesting to investigate approaches of dynamically (un)loading software components for mobile applications, in which issues such as memory usage are extremely important to be taken into account.

Instantiating our Framework. The main features of the current version of our framework are: (i) it has the infrastructure needed to instantiate and persist the User metamodel we proposed in [Nunes et al. 2010]; and (ii) it provides an adaptation mechanism to evolve the system to be consistent with the User Model. These are the main tasks that are necessary to instantiate the framework: (a) choose the database to persist models; (b) define and implement the Domain Model; (c) define Variability and Preferences Definition Models. These models are directly persisted in the database; (d) implement User Agents using variability implementation techniques; (e) define adaptation rules and actions. These are defined in a Spring configuration file.

Our framework was built using an approach in which the domain (personal assistance software) was extensively analyzed, and then the framework with its frozen and hot-spots was designed and implemented. We have instantiated our framework for an application that provides traveling services.

5. Related Work

Most of the works related to agent-based personal assistance software address eliciting and reasoning about user preferences. They focus on the domain of recommender systems and *which* are the user preferences, so that the system is able to provide a recommendation of one or a subset of a set of choices. So they do not address the impact of preferences on the system behavior and *how* they are realized, which is the goal of our work. Next we introduce and discuss work related to this issue.

A multi-agent infrastructure, named Seta2000, for developing personalized web-based systems is presented in [Ardissono, L. et al. 2005]. It supports the development of recommender systems by the provision of two main behaviors: personalized suggestion of items, and the dynamic generation of user interfaces. The main contribution from a personalization perspective of Seta2000 is the reusable recommendation engine that can be customized to different application domains. Even though this work provided a reusable infrastructure to build web-based recommender systems, it did not provide new solutions in the context of personalized systems: it leverages existing recommendation techniques and provides an extensible implemented agent-based solution. This can be seen in the related work reported in [Ardissono, L. et al. 2005], in which Seta2000 is compared to general purpose agent platforms, such as JADE.

Huang et al. [Huang et al. 2008] describes a personalized recommendation system based on multi-agents. The system provides an implicit user preferences learning approach, and distributes responsibilities of the recommendation process among different agents, such as learning, selection & recommendation and information collection agent. These agents are an underlying infrastructure of an intelligent user agent. As the previous discussed work, this system adopts existing techniques to build recommendation systems.

One of the biggest projects in the context of personalized user agents is the Cognitive Assistant that Learns and Organizes (CALO) project[4] [Berry et al. 2006, Berry et al. 2009], whose goal is to support a busy knowledge worker in dealing with the twin problems of information and task overload. Along the project, the research effort was mostly concentrated in the PTIME agent, which is an autonomous entity that works with its user, other PTIME agents, and other users, to schedule meetings and commitments in its user's calendar. One main issue of this work is that the solution is highly coupled with the domain being explored (meetings scheduling). In addition, as there is only one concern being addressed by PTIME (scheduling), there are not multiple concerns related to user customizations, which is a problem we are looking at. Furthermore, this research work did not address user configurations, i.e. the system that all users are managing are not tailored for their needs in the sense of features that the system provides. Despite this limitations, the CALO project substantially advanced on the development of user agents, also taking into account human-computer interaction (HCI) issues that are essential to improve the chances of users adopting personal agents. Therefore, lessons learned from this project [Berry et al. 2009] can be leveraged in our work.

6. Conclusion

We presented a framework for developing personal assistance software, which follows a reference architecture previously proposed for this application domain. Our framework provides the implementation of a user metamodel, which captures user configurations and preferences in a high-level way. The implementation also provides means for persisting instantiated models. In addition, the framework aggregates interface components that allows manipulating user models in a graphical manner. Moreover, our framework includes a mechanism for adapting applications at implementation level based on changes on the user model. This mechanism uses as input adaptation rules and actions, which are executed when the changes on the user model comprises the context of the rule application.

[4]http://caloproject.sri.com/

Providing an infrastructure that uses a high-level user model to drive adaptations on personal assistance software has several advantages: (i) user customizations are implementation-independent; (ii) the user model vocabulary becomes a common language for users specifying configurations and preference; (iii) the user model modularizes customizations, allowing a modular reasoning about them. Future work includes addressing modules (learning and security) currently not covered by our framework.

Acknowledgments

This work has been 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 - G2. Define a framework to support the development of autonomic Web applications.

References

Ardissono, L. et al. (2005). A multi-agent infrastructure for developing personalized web-based systems. *ACM Trans. Internet Technol.*, 5(1):47–69.

Berry, P., Peintner, B., Conley, K., Gervasio, M., Uribe, T., and Yorke-Smith, N. (2006). Deploying a personalized time management agent. In *AAMAS '06*, pages 1564–1571.

Berry, P. M., Donneau-Golencer, T., Duong, K., Gervasio, M., Peintner, B., and Yorke-Smith, N. (2009). Evaluating user-adaptive systems: Lessons from experiences with a personalized meeting scheduling assistant. In *IAAI'09*, pages 40–46.

Huang, L., Dai, L., Wei, Y., and Huang, M. (2008). A personalized recommendation system based on multi-agent. In *WGEC '08*.

Kästner, C. and Apel, S. (2009). Virtual separation of concerns - a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78.

Nunes, I. (2010). A bdi extension for jade. http://www.inf.puc-rio.br/~ionunes/bdi4jade/.

Nunes, I., Barbosa, S., and Lucena, C. (2010). An end-user domain-specific model to drive dynamic user agents adaptations. In *SEKE 2010*, USA.

Nunes, I., Lucena, C. J., Cowan, D., and Alencar, P. (2009). Building service-oriented user agents using a software product line approach. In *ICSR '09*, pages 236–245.

Pohl, K., Böckle, G., and van der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag.

Rao, A. and Georgeff, M. (1995). BDI-agents: from theory to practice. In *ICMAS'95*.

Rogoff, K. (2010). Grandmasters and global growth. *Project Syndicate*. Available at http://www.project-syndicate.org/commentary/rogoff64/English.

Shaw, M. and Clements, P. (2006). The golden age of software architecture. *IEEE Softw.*, 23(2):31–39.