Increasing Users' Trust on Personal Assistance Software using a Domain-neutral High-level User Model

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

PUC-Rio, Computer Science Department, LES - Rio de Janeiro, Brazil
{ionunes,simone,lucena}@inf.puc-rio.br

Abstract. People delegate tasks only if they trust the one that is going to execute them, who can be a person or a system. Current approaches mostly focus on creating methods (elicitation approaches or learning algorithms) that aim at increasing the accuracy of (internal) user models. However, the existence of a chance of a method giving a wrong answer decreases users' trust on software systems, thus preventing the task delegation. We aim at increasing users' trust on personal assistance software based on agents by exposing a high-level user model to users, which brings two main advantages: (i) users are able to understand and verify how the system is modeling them (transparency); and (ii) it empowers users to control and make adjustments on their agents. This paper focuses on describing a domain-neutral user metamodel, which allows instantiating high-level user models with configurations and preferences. In addition, we present a two-level software architecture that supports the development of systems with high-level user models and a mechanism that keeps this model consistent with the underlying implementation.

1 Introduction

As web applications become increasingly interactive, accessible, and pervasive the web is providing mechanisms that can help users extend their mental and physical capabilities. The Web now provides access to huge amounts of well-organized information and supports social interactions well beyond our physical limitations. Thus there are new challenges in managing both the quantity of information and the complexity and timeliness of relationships. Multi-agent Systems (MASs) [1], with roots not only in Artificial Intelligence (AI) but also in distributed systems and software engineering, can incorporate autonomous behavior to support web users in meeting many of these new barriers by freeing users from repetitive and tedious tasks. In addition, MASs, by providing autonomous behavior, may be employed by web users to support access to information and decision-making. The concept of user (or personal) agents was championed by Maes in 1994. In [2], she introduced the idea that autonomous agents may be personal assistants who are collaborating with the user in the same work environment. However, even though significant research effort has been invested on developing user agents, we are far from their massive adoption.

People delegate tasks only if they trust the one that is going to execute them, who can be a person or a system. For user agents, this claim is supported by the study presented by Schiaffino & Amandi in [3]. Their study showed that users fear having a completely automated agent. In addition, they concluded that a large group of users is willing to adopt user agents only if they know exactly what the agent is going to do, i.e. if they trust that the agent will perform a behavior previously approved by them. Current approaches mostly focus on creating methods (elicitation approaches or learning algorithms) that aim at increasing the accuracy of (internal) user models. Nevertheless, the risk that a method gives a wrong answer decreases users' trust on software systems, thus preventing the task delegation.

Our research aims at increasing the level of acceptance of user agents (and task delegation to computer systems) by users. In order to achieve this our goal is to increase users' trust on personal assistance software based on two main properties: (i) transparency; and (ii) power of control. The main idea is to expose user models (or profiles) in an end-user-readable manner, and therefore users can understand the system behavior (transparency). In addition, users can manage their model to control the agents' behavior (power of control).

In this paper we demonstrate steps in creating a model-driven approach for developing personal assistance software based on high-level user models and user agents. We present an approach to empower users with a high-level domain-specific language that allows them to dynamically program and personalize their agents. Even though inference models might reach the wrong conclusions about user preferences and cause agents to take inappropriate actions, they can be leveraged to create initial versions of the user model, so that users can make fine-grained modification on it. The steps to be presented are: (i) a high-level user metamodel to represent user configurations and preferences; and (ii) a software architecture to build personal assistance software based on agents that are adapted driven by instances of the proposed metamodel.

The proposed Domain-specific Model (DSM) (high-level user metamodel) provides the necessary vocabulary to build an end-user configurations and preferences language. Existing representation models of user preferences force users to express their preferences in a particular way. Consequently, these works create the need for elicitation techniques to interpret answers to questions and indirectly build the user model. The language that our DSM creates allows users to configure their agents and to express different kinds of preference statements, creating a vocabulary that allows representing statements close to the ones expressed in natural language. The proposed DSM is an application-domain-neutral metamodel that may be instantiated to build different applications.

The paper also describes a software architecture to build personal assistance software that follows our approach. The architecture is composed of user agents that are dynamically adapted based on a user model that follows our metamodel. In this sense, users' customizations are represented in a high-level user model and realized at the implementation-level by user agents. As configurations and preferences may impact in different agents and their components, the user model becomes a modularized view of users' customizations, facilitating their management, considering that they change over time. However, given that there are users' customizations represented in two different levels of abstractions, we also present an algorithm that keeps both representations consistent.

This paper is organized as follows. Section 2 describes our metamodel. In Section 3, we describe the software architecture for building personal assistance software. Section 4 presents an evaluation of our metamodel by showing its generality when used across different domains. Section 5 presents related work. Finally, Section 6 concludes this paper.

2 A Domain-neutral User Metamodel

One of the most challenging tasks in building personal assistance software able to act on users' behalf is to capture particularities of the person being represented (user model) so that the system can present an appropriate behavior. We aim at exposing the user model to users in order to provide them the power of controlling their agents and increasing the trust on the system. This user model must be expressed with very high-level abstractions, otherwise users are not able to understand them. This section presents our proposal of a domain-neutral high-level user metamodel to represent users' customizations.

We acknowledge the relevance of the existence of a reasoning algorithm for user models. However, our goal is to use the proposed metamodel in a level higher than the ones processed by algorithms. When users inform their preferences to applications with restricted user models, e.g. models with boolean preferences, they have to translate preferences statements expressed in natural language to the one imposed by the application. Our goal with this high-level model is to provide users with a larger vocabulary to express their preferences and leave the task of translating preferences for a particular model to the system, as Figure 1 illustrates. For instance, a user may say: "My preference is not to buy a laptop of the brand X, but I don't care if the brand is A, B or C." Suppose now that the system uses a reasoning algorithm to process preferences expressed as a partial order relation. We can represent that preference like this:

$$preference = \{< A, X >, < B, X >, < C, X >\}.$$

Therefore, our metamodel is implementation-independent.

It is important to highlight that our user metamodel 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. They are represented by optional and alternative features that users choose for customizing their application. A feature, in turn, is any variable characteristic of the system. On the other hand, 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 user rates certain options

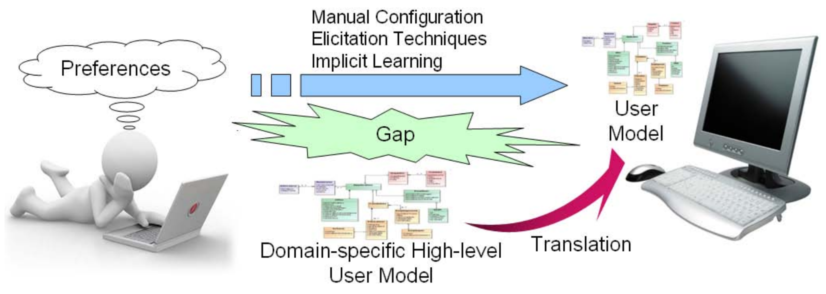


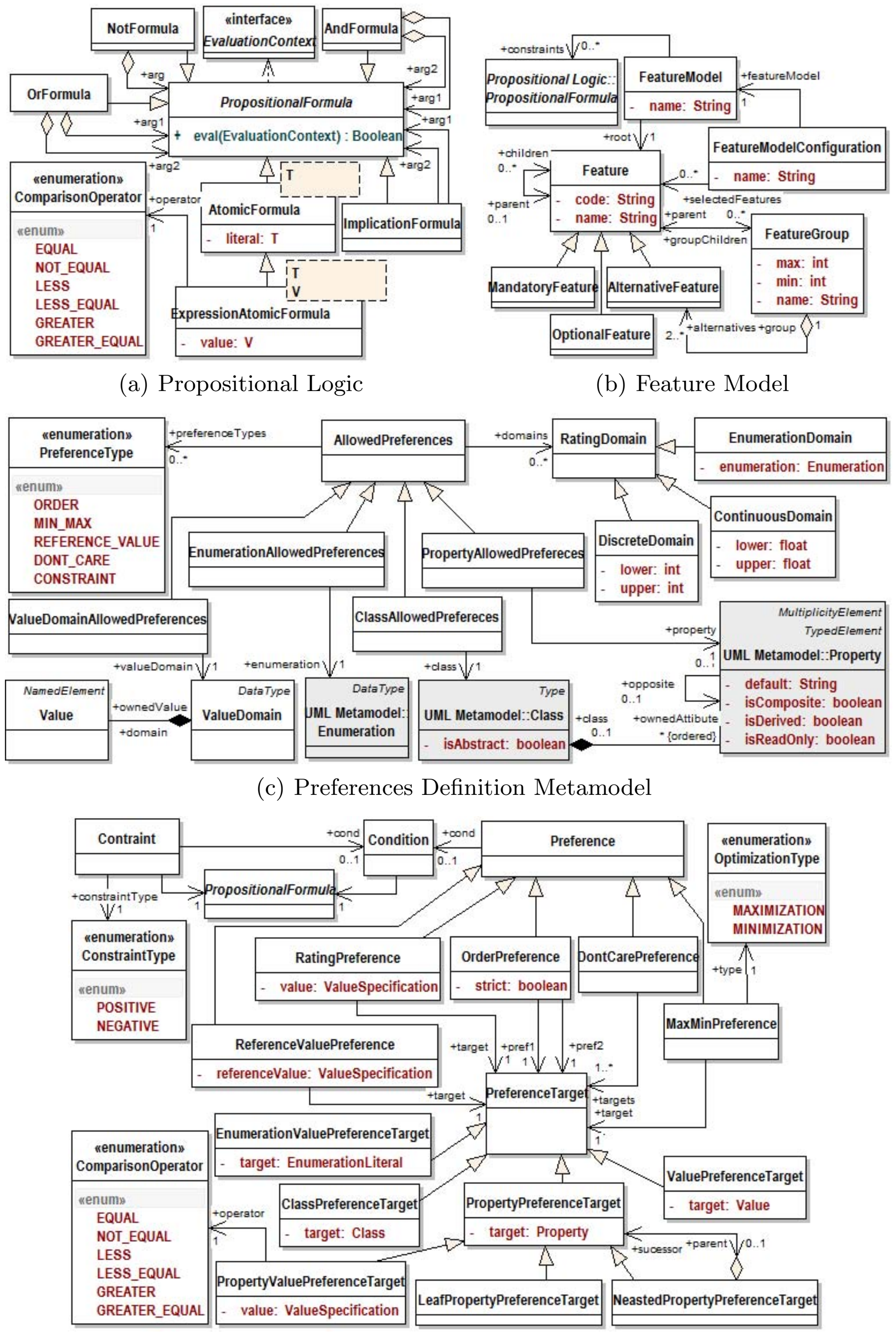
Fig. 1. High-level User Model.

better than others in certain contexts. They are a representation of the cognitive model of the user in order for agents to behave and make decisions in a way as close as possible as users would do.

Our user metamodel is instantiated in a stepwise fashion. First, application developers instantiate part of the metamodel for defining application-specific abstractions and constraints. This is performed at development time. Second, at runtime, the user instantiates preferences and configurations in order to customize the personal assistant application. Our metamodel, which is an extension of the UML metamodel[1], is depicted in Figure 2 in four different parts. Elements of the UML metamodel, e.g. `Class` and `Property`, are either distinguished with a gray color in diagrams or are referred to in properties. Next we describe our metamodel more extensively.

There are three models that must be instantiated at development time: (i) Ontology model; (ii) Feature model; and (iii) Preferences Definition model. The Ontology model represents the set of concepts within the domain and the relationships between those concepts. The Feature model (Figure 2(b)), in turn, allows modeling variable traits within the domain, which are later used for defining user configurations. This model incorporates the ideas of Software Product Lines (SPLs) [4] and their feature models [5]. SPL is a new software reuse approach that aims at systematically deriving families of applications based on a reusable infrastructure with the intention of achieving both reduced costs and reduced time-to-market. The goal of the Feature model is to describe variation points and variants in the system, which can be either optional or alternative, and can be added and removed dynamically from the application. A `FeatureModel` is a tree of `Feature`s. A `Feature` can be mandatory, optional and alternative. Mandatory features are represented only if they are in a chain with other optional and alternative features, and therefore need to be represented. Otherwise, they will be present in the system any way. `AlternativeFeature`s are grouped into `FeatureGroup`s, which define the minimum and maximum number of alternatives that can be chosen. Finally, constraints may be defined in order to represent relationships between variations, e.g. a feature requires the presence

[1] http://www.omg.org/spec/UML/



(a) Propositional Logic

(b) Feature Model

(c) Preferences Definition Metamodel

(d) Preferences Metamodel

Fig. 2. User Metamodel.

of another. These constraints are expressed as `PropositionalFormula`s, shown in Figure 2(a).

The goal of our metamodel is to allow users not only to customize their applications with selected features, as in a SPL, but also to capture users' preferences in order for automated customized agents to act appropriately on their behalf. The Preferences Definition model defines restrictions over preferences that users can express. Its metamodel is presented in Figure 2(c). The purpose of this model is to define *how* users can express their preferences and *about which elements* of the Ontology Model. 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 if there is no mechanism to translate them to quantitative statements.

Users can express different types of preference: (i) Order (`ORDER`) – expresses an order relation between two elements, allowing users to express *"I prefer trains to airplanes."* A set of instances of the Order preference comprises a partial order; (ii) Reference Value (`REFERENCE_VALUE`) – 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; (iii) Minimize/Maximaze (`MIN_MAX`) – indicates that the user preference is to minimize or maximize a certain element; (iv) Don't Care (`DONT_CARE`) – it allows indicating a set of elements that users do not care about. It is useful for users to express *"I don't care if I travel with company A, B or C;"* (v) 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, e.g. LOVE, LIKE, INDIFFERENT, DISLIKE and HATE. Moreover, different domains can be specified for the same element. Using Rating preferences, it is possible to assign utility values to elements, or to express preference statements; and (vi) Constraint (`CONSTRAINT`) – a particular preference type that establishes a hard constraint over decisions, as opposed to the other preference types, used to specify soft constraints. Constraints allows users to express strong statements, e.g. *"I don't travel with company D."*

Different kinds of preferences may be used by agents in different ways, according to the approaches they are using to reason about preferences. If an agent uses utility functions and the user defines that the storage capacity of a computer must be maximized and provides a reference value α, the agent may choose a utility function like $f(x) = \sqrt[\alpha]{x}$.

For defining the allowed preference types, developers must create instances of `AllowedPreferences`, and make the corresponding associations with types and domains. The specializations of `AllowedPreferences` characterize different element types that can be used in preference statements. There are four different possibilities: classes (*I prefer notebook to desktop*), properties (*The notebook weight is an essential characteristic for me*) and their values (*I don't* *like notebooks whose color is pink*), enumeration literals (*I prefer red to blue*) and values (*Cost is more relevant than quality*). Value is a first-class abstraction that we use to model high-level user preferences. Values are essential when using a value-focused thinking [6]: *"Values are what we care about. As such, values should be the driving force for our decision making. They should be the basis for the time and effort we spend thinking about decisions."* A scenario that illustrates the use of values is in the travel domain. A user may have comfort (a value) as a preference when choosing a transportation, instead of specifying fine-grained preferences, such as *trains are preferred to airplanes*, but *traveling in an airplane first-class is better than by train*, and so on. In this case, the user agent is a domain expert that knows what comfort means.

Based on these three instantiated models and on our Preferences metamodel (Figure 2(d)), it is possible to build a User Model to model preferences and configurations. It is composed of two parts: (i) Configuration model; and (ii) Preferences model. As discussed above, in the Configuration model, users choose optional and alternative features (variation points) from the Feature model, defining their configurations, which are instance of the `FeatureModelConfiguration`. Therefore, a configuration is a valid set of selected optional and alternative features of a `FeatureModel`. On the other hand, in the Preferences model, users define a set of preferences and a set of constraints. These are more closely related to a cognitive model of the user. User preferences (or soft constraints) determine what the user prefers, and indirectly how the system *should* behave. If the preferred behavior is not possible, the system may move to other acceptable alternatives. Constraints, in turn, are restrictions (hard constraints) over elements. As opposed to preferences, they directly define mandatory or forbidden choices that *must* be respected by the system.

Figure 2(d) shows the `Constraint` element and five different specializations of `Preference` that represent the different preference types previously introduced. Constraints are expressed in propositional logic formulae, however using only $\neg$, $\wedge$ and $\vee$ logical operators. Atomic formulae refer to the same types of elements of preferences and can use comparison operators ($=$, $\neq$, $>$, $\geq$, $<$, $\leq$) between properties and their values. The `PreferenceTarget` and its subtypes are used to specify the element that is the target of the preference statement or formula. In addition, it allows to specify nested properties, such us `Flight.arrivalAirport.location.country`. If we have directly associated preferences to classes, properties, enumerations and values, either we would have to make specializations of each preference type to each element type or to change the UML metamodel to make a common superclass of classes, properties, enumerations and values. Given that we did not want to modify the UML metamodel, but only to extend it, and the first solution would generate four specializations for each preference type, we used the `PreferenceTarget` as an indirection for elements that are referred in preferences and constraints.

Besides defining preferences and constraints, users can specify conditions, also expressed in propositional logic formulae (Figure 2(a)), to define contexts in which preferences and constraints hold. Furthermore, in order to guarantee that users produce valid instances of the metamodel, we have defined additional constraints over instantiated models, e.g. in a nested property, the child of a property whose class is X must also be a property of Class X.

3 A Two-level Software Architecture for Building Personal Assistance Software

The main contribution of this paper is the user metamodel described previously. However, we also present a software architecture that provides a structure of modules and incorporates the idea of a high-level user model. This software architecture addresses the domain of personal assistance software. The structure of this architecture aims at building systems composed of personalized user agents which are adapted based on user models, which can be instantiated and modified by end-users.

The goal of the proposed architecture is to accommodate a high-level user model, but also to allow building high quality personal assistance software by taking into account good software engineering practices. User customizations may be seen as a *concern* in a system that is spread all over the code. However, at the same time, each customization is associated with different services (also concerns) provided to users. Therefore, when developing such system one has to choose the dimension in which the software architecture will be modularized: in terms of services (Figure 3(a)) or modularizing user settings in a single model (Figure 3(b)). It can be seen that it is not possible in either approach to modularize concerns in single modules. In addition, without modularizing user customizations, as in Figure 3(a), they are buried inside the code, thus making it difficult to understand them as a whole.

Moreover, user preferences play different roles in agent architectures [7, 8]. We illustrate examples of these roles in Table 1. If all this information is contained in a single user model, we have the problems discussed above and this model would aggregate information related to different concerns of the system (low cohesion among user model elements).

Our solution is to provide a *virtual separation of concerns* [9]. A concern is anything that is interesting from the point of view of a stakeholder. In our case, the concern that will be virtually modularized is the user model. The main idea is to structure the user agent architecture in terms of services by modularizing

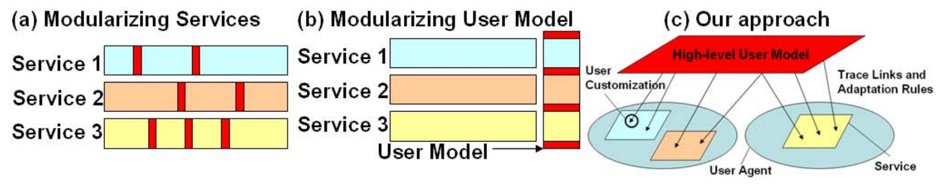


Fig. 3. Modularization Approaches.

Attitude	**Example**
Goal	I want to drink red wine.
Belief	I like red wine.
Motivation	Red wine is good for the heart.
Plan	In order to drink red wine either I go to the supermarket and buy a bottle (plan A) or I go to my friend's home who always have wine there (plan B).
Meta-goal	I want to drink red wine, but spending less money as possible (so I might choose plan B).

Table 1. User Preferences and their roles into agent architectures.

its variability as much as possible into agent abstractions. We provide a virtual modularized view of user customizations, as Figure 3(c) illustrates. Customizations are not design abstractions, but they are implemented by typical agent abstractions (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, and users are able to configure their agents by means of this model. Using a high-level user model to drive adaptations on personal assistance software brings the main following advantages: (i) user customizations are implementation-independent; (ii) the vocabulary used in the user model becomes a common language for users specifying configurations and preference; (iii) the user model modularizes customizations, allowing a modular reasoning about them.

3.1 Detailing our Software Architecture

In this section we detail our proposed architecture, depicted in Figure 4, and describe the mechanism that makes the high-level user model (henceforth referred to as user model) work with agent architectures.

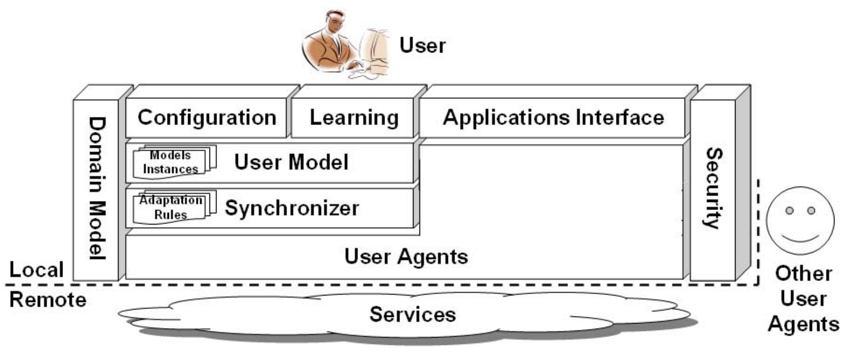


Fig. 4. Proposed Architecture.

The *User Agents* module 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 to reason about preferences. We propose to adopt an agent-based approach to design and implement user-customizable systems for 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 between what to do (goals) and how to do it (plans).

User Agents use services provided by a distributed environment (the *Services* cloud), and their knowledge is based on the *Domain Model*, composed of entities shared by user agents and services, application-specific, etc. The *Security* module addresses security and privacy issues, because user agents may share information with other user agents. This module aggregates policies that restrict this communication, assuring that confidential information is kept safely secured. Users access services provided by user agents through the *Applications Interface* module.

The *User Model* contains user configurations and preferences expressed in a high-level language. They are present in the user agents architecture but as design-level abstractions. 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. In addition, 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*. This module has a degree of autonomy parameter, so it may automatically change the *User Model*, or just suggest changes to it, to be approved by the end users.

The *User Model* and *User Agents* are connected by representing user customization in two different levels of abstraction. This connection is stored in the form of trace links, indicating how and where a customization is implemented in a user agent(s). Adaptations are performed at runtime and are accomplished based on the trace links between the *User Model* and the *User Agents* architecture. The *Synchronizer* is the module in charge of adapting *User Agents* based on changes in the *User Model*. It is able to understand these trace links, and it knows which transformation must be performed in the *User Agents* based on changes in the *User Model*. Therefore, the *User Model* drives adaptations in the *User Agents*.

The algorithm we define to be performed by the *Synchronizer* module is presented in Algorithm 1. It receives as input a previous and an updated versions of a user model, as well as a map containing rules that states which set of actions must be performed when an event (a change) on the user model occurs. The algorithm first calculates the set of events (changes) between the two versions of the user model (line 1). Next, for each event, it gets the set of rules that "listen" to the event (line 2–6). The result is the set of rules that produce actions for the event set. Then, it is retrieved, from each rule, the set of actions (changes) that must be performed at the implementation level of the system (lines 7–9), so that it turns to be consistent with the updated version of the user model. Finally, all actions are performed (lines 10–11).

Algorithm 1: Adaptation algorithm

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

Examples of actions are the addition or removal of agents, beliefs, goals and plans. Events are the (de)selection of features or addition or removal of a preference over a certain element. And rules associate events with actions.

4 Instantiating our User Metamodel for Different Application Domains

Our metamodel was built using preference statements collected from different individuals in a user study and from papers related to user preferences. The idea was to contemplate the different kinds of preference statements in order to maximize the users' expressiveness. The metamodel uses abstractions from the user preferences domain, therefore the language is built as an end-user language. This section presents two Preferences models to show that our metamodel is generic enough to model different kinds of preferences statements in different domains – flight reservation and computer purchase domains. Given that these are two well-known domains, we assume that the reader is familiar with them, and due to space restrictions, we present only the Preferences models. In addition, we assume that the Preferences Definition model defines that all preference types over all elements are allowed.

The first Preference model, which is from the flight domain, indicates where a user prefers to sit inside an airplane. This model consists of three order preferences, two of them with conditions, and one minimization preference. Next, we present the four modeled preference statements in natural language, and Figure 5 shows how they are modeled with our metamodel abstractions.

P1. *If the flight is short, i.e. its duration does not exceed 4 hours, I prefer a seat by the aisle to a seat by the window.*
P2. *If the flight is long, i.e. its duration is higher than 4 hours, I prefer a seat by the window to a seat by the aisle.*
P3. *I always prefer to sit at the first rows of the airplane.*
P4. *Sitting at the first rows of the airplane is more important to me than the seat location.*

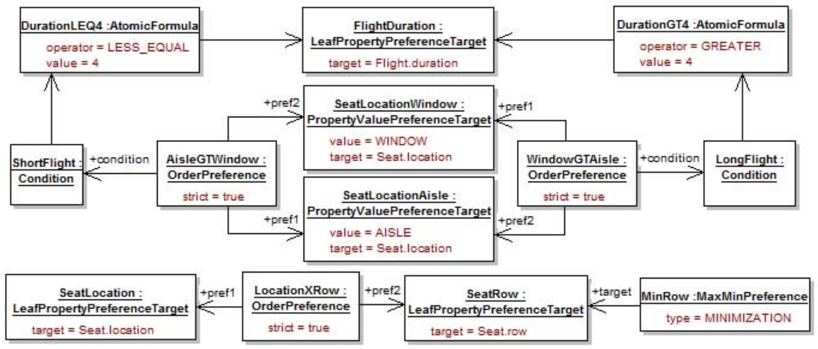


Fig. 5. User Preferences model in Flight Domain.

The computer domain Preferences model presented in Figure 6 has some elements in gray color. They are not part of the Preferences model, but from the Domain model, but we included them in Figure 6 to present some application-specific concepts used in this model. First, four values are defined in the Computer Domain (mobility, readability, performance and cost). These values can be rated with "+", ranging from one to five. These are the natural language preference statements modeled in Figure 6:

P1. *Cost is the most important value (+++++).*
P2. *I rate performance with ++++.*
P3. *I rate readability with ++++.*
P4. *I rate mobility with ++.*
P5. *I'm expecting to pay around $800 for my laptop.*
P6. *I want a computer with less than 3Kg.*
P7. *The lighter the computer is, the better.*

It is important to notice that Rating and Order preferences provide different information. By saying that cost is +++++ and performance is ++++, a user is informing that cost is more important than performance (order), but performance is also important, and should be taken into account.

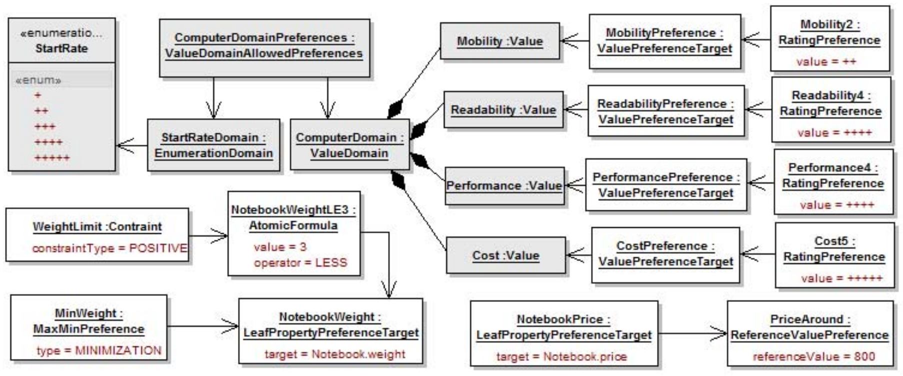

«enumeratio... StartRate
«enum»
+
++
+++
++++
+++++
ComputerDomainPreferences : ValueDomainAllowedPreferences
StartRateDomain : EnumerationDomain
ComputerDomain : ValueDomain
Mobility :Value
MobilityPreference : ValuePreferenceTarget
Mobility2 : RatingPreference
value = ++
Readability :Value
ReadabilityPreference : ValuePreferenceTarget
Readability4 : RatingPreference
value = ++++
Performance :Value
PerformancePreference : ValuePreferenceTarget
Performance4 : RatingPreference
value = ++++
Cost :Value
CostPreference : ValuePreferenceTarget
Cost5 : RatingPreference
value = +++++
WeightLimit :Contraint
constraintType = POSITIVE
NotebookWeightLE3 : AtomicFormula
value = 3
operator = LESS
MinWeight : MaxMinPreference
type = MINIMIZATION
NotebookWeight : LeafPropertyPreferenceTarget
target = Notebook.weight
NotebookPrice : LeafPropertyPreferenceTarget
target = Notebook.price
PriceAround : ReferenceValuePreference
referenceValue = 800

Fig. 6. User Preferences model in Computer Domain.

5 Related Work

Several approaches have been proposed to deal with user preferences. To build our metamodel, we have conducted extensive research on which kinds of preferences other proposals represent and additional concepts they define. Typically, preferences are classified as quantitative or qualitative (e.g. "I love summer" versus "I like winter more than summer"). Both approaches can be represented through our metamodel. Quantitative preferences are modeled in the framework proposed in [10] by means of a preference function that maps records to a score from 0 to 1. On the other hand, CP-Nets [11] models qualitative preferences. CP-Nets also allow modeling conditionality, which is considered in our work as well. The concept of normality is defined in [12], so that users can express preferences considering normal states of the world, but these preferences may change when the world changes. The normality abstraction can be modeled using conditions in our metamodel.

Ayres & Furtado proposed the OWLPref [13], a declarative and domain-independent preference representation in OWL. This work has the same purpose of our work in the sense that it generically models user preferences. However, OWLPref does not precisely define the preferences model, e.g. lacking the definition of associations, it shows only a hierarchical structure of preferences. A preference metamodel is also proposed in [14]. However, its expressiveness is very limited. It only allows to define desired values (or intervals) of object properties.

One of the biggest projects in the context of personalized user agents is the Cognitive Assistant that Learns and Organizes (CALO) project[2] [15, 16], 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. Users are able to express their

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

preferences, nevertheless the adopted language is tight to application domain (meeting scheduling). Despite this limitation, the CALO project substantially advanced on the development of user agents, also taking into account human-computer interaction (HCI) issues that are essential for improving the chances of users adopting personal agents. Therefore, lessons learned from this project [16] can be leveraged in our work.

6 Conclusion

With the growth of the Internet, interactivity and access to information are significantly increasing. At the same time, several of our everyday-tasks are being managed by software applications, such as to-do lists and schedules. The combination of these trends converge to the automation of user tasks performed by agents that act on behalf of users. Agents must have reliable user models that assure they act appropriately, otherwise they will not be trusted by users.

In order to increase users' trust on personal assistance software based on automated agents, we proposed in this paper the idea of exposing high-level user models to users so that they can verify and understand this model as well as control it by configuring it and making fine-grained modifications. Our proposal is a domain-specific metamodel that provides abstractions from the user domain, including configurations, constraints and preferences. Different abstractions used by end users in natural language statements are directly represented. Users are able to tailor personal assistance software systems with optional and alternative features, and model their preferences. Besides (hard-)constraints, five different preferences types (soft-constraints) can be represented: order, rating, reference value, maximization/minimization and don't care. In addition, we adopt values as a first-class abstraction to model high-level preferences. Instances of our metamodel are to be used in combination with our proposed software architecture, which uses them as a global view of user customizations. Services are provided by user agents structured with traditional agent-based architectures. The User Model provides a modularized view of different user-related concepts spread into agent architectures. We also presented an algorithm to be performed by the Synchronizer module, which ensures that changes in the User Model demands appropriate adaptations in user agents.

We are currently working in several directions. First, we made a survey of preference statements provided by user in order to build a language based on our metamodel using syntactic sugar. In addition, we are investigating how to verify the User Model to identify inconsistencies across preferences. Finally, we are implementing a framework based on our software architecture to provide a solid infrastructure to build personal assistance software systems.

Acknowledgements

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 - G1. Design techniques to improve the development of autonomic Web applications. Ingrid Nunes #141278/2009-9, Simone Barbosa #313031/2009-6, Carlos Lucena #304810/2009-6 also thank CNPq for respective research grants.

References

1. Weiss, G., ed.: Multiagent systems: a modern approach to distributed artificial intelligence. MIT Press, Cambridge, MA, USA (1999)
2. Maes, P.: Agents that reduce work and information overload. Commun. ACM **37**(7) (1994) 30–40
3. Schiaffino, S., Amandi, A.: User - interface agent interaction: personalization issues. Int. J. Hum.-Comput. Stud. **60**(1) (2004) 129–148
4. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag (2005)
5. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, SEI (1990)
6. Keeney, R.L.: Value-focused thinking – A Path to Creative Decisionmaking. Harvard University Press, London, England (1944)
7. Doyle, J.: Prospects for preferences. Computational Intelligence **20** (2004) 111–136
8. Nunes, I., Barbosa, S., Lucena, C.: Modeling user preferences into agent architectures: a survey. Technical Report 25/09, PUC-Rio, Brazil (September 2009)
9. Kästner, C., Apel, S.: Virtual separation of concerns - a second chance for preprocessors. Journal of Object Technology **8**(6) (2009) 59–78
10. Agrawal, R., Wimmers, E.L.: A framework for expressing and combining preferences. In: 2000 ACM SIGMOD. (2000) 297–306
11. Boutilier, C., Brafman, R.I., Hoos, H.H., Poole, D.: Cp-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. Journal of Artificial Intelligence Research **21** (2004) 135–191
12. Lang, J., van der Torre, L.: From belief change to preference change. In: ECAI 2008, The Netherlands, IOS Press (2008) 351–355
13. Ayres, L., Furtado, V.: Owlpref: Uma representação declarativa de preferências para web semântica. In: XXVII Congresso da SBC, Brazil (2007) 1411–1419
14. Tapucu, D., Can, O., Bursa, O., Unalir, M.O.: Metamodeling approach to preference management in the semantic web. In: M-PREF 2008, USA (2008) 116–123
15. Berry, P., Peintner, B., Conley, K., Gervasio, M., Uribe, T., Yorke-Smith, N.: Deploying a personalized time management agent. In: AAMAS '06. (2006) 1564–1571
16. 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. (2009) 40–46