University of Waterloo

Technical Report
CS-2010-05

An End-user Domain Specific Model to Drive Dynamic User Agents Adaptations

Ingrid Oliveira de Nunes
Simone Diniz Junqueira Barbosa
Carlos José Pereira de Lucena

Faculty of Mathematics

DAVID R. CHERITON SCHOOL OF COMPUTER SCIENCE
UNIVERSITY OF WATERLOO
WATERLOO, ONTARIO, CANADA N2L 3G1

Technical Report CS-2010-05 March, 2010
David R. Cheriton School of Computer Science - University of Waterloo

An End-user Domain Specific Model to Drive Dynamic User Agents Adaptations

Ingrid Oliveira de Nunes[1], **Simone Diniz Junqueira Barbosa**[1], **Carlos José Pereira de Lucena**[1,2]

[1] Pontifical Catholic University of Rio de Janeiro (PUC-Rio) - Rio de Janeiro, Brazil
[2] University of Waterloo - Waterloo, Canada

{ionunes,simone,lucena}@inf.puc-rio.br

Abstract. Modeling automated user tasks based on agent-oriented approaches is a promising but challenging task. Personalized user agents have been investigated as a potential way of addressing this issue. Most of recent research work has focused on learning, eliciting and reasoning about user preferences and profiles. In this paper, our goal is to deal with the engineering of such systems, barely discussed in the literature. In this context, we present a high-level domain specific model whose aim is to give users the power of customizing and dynamically adapting their user agents. In addition, we propose a general architecture to develop agent-based systems able to be customized to users.

Keywords: User Agents, Domain Specific Modeling, Adaptation, Personalization, Multi-agent Systems.

In charge of publications:
Helen Jardine
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
Tel: +1 519 888-4567 x33293 Fax: +1 519 885-1208
E-mail: hjardine@uwaterloo.ca
Web site: http://www.cs.uwaterloo.ca/research/tr/

1 Introduction

Recently, Rogoff discussed the envision that the next big driver of global growth is the Artificial Intelligence (AI) area [15]. He claims that computers are going to automatically perform a growing range of tasks in the next 50 years. Multi-agent Systems (MASs) [19], with roots not only in AI but also in distributed systems and software engineering, have addressed this domain of applications, 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 an agent to meet specific needs of the users [13]. The concept of user (or personal) agents were championed by Maes in 1994. In [11], she discusses the large number of tasks that emerge from the use of computers and the web, and autonomous agents may be personal assistants who are collaborating with the user in the same work environment.

Several research work has been carried out in the context of user agents. It includes learning preferences by monitoring users [16], eliciting preferences by interacting with them [10] and reasoning about preferences [4]. Even though these works significantly advanced the area of user agents, few 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 [17].

Our research addresses software engineering issues related to the development of user agents, considering the need of personalizing them to individual users. In previous work [13], we have proposed an approach for building customized service-oriented user agents. The main idea was to capture the domain variability into a variability model, and to develop an agent product line [14] that supports this identified variability. A user is able to choose a configuration of the variability model, and then a customized agent is deployed into a MAS to provide a service for the user. The present work has two significant differences from [13]: (i) our goal is to deal with dynamic adaptations, i.e. user agents may evolve at runtime; and (ii) in [13], we consider only *configurations* and now we are also considering *preferences*. By configuration, we mean a setting that a user performs in a system, such as by adding or removing a service or enabling an optional feature. These configurations can be related with environment restrictions, e.g. a device configuration. Preferences, in turn, are how users consider an option better than another. Such information is typically used in the agent reasoning process. Both concepts are part of the user model, and together we refer to as *customizations*.

To deal with dynamic adaptations, we propose a software architecture to build user-driven applications. The core of this architecture are user agents that provide customized services for users. Other modules of the architecture give additional infrastructure, such as security, and support the adaptation process. User agents are personalized based on an end-user Domain Specific Model (DSM) that allows to model user customizations, and they are adapted according to changes in this model. Besides describing our proposed software architecture, we also focus on this DSM in this paper. We describe a meta-model, which empowers end-users to program their agents in specific applications. Users typically do not fill extensive forms on the web to receive recommendations of products, but they may spend time configuring a system if they need some tedious or repetitive tasks to be accomplished. Moreover, current algorithms that capture user models can easily remember user interactions, however it takes a lot of time until they forget them, and this may bother the user with completely irrelevant information. Our proposed DSM allows the user

to make such changes.

The remainder of this paper is organized as follows. Sections 2 and 3 present the main contribution of this paper, detailing our proposed software architecture and the meta-model, respectively. Section 4 evaluates our approach, showing its generality when used across different domains. Section 5 presents related work, followed by Section 6, which concludes this paper.

2 A User-driven Software Architecture

Our goal is to build adaptive user agents. In particular, we aim at addressing two main issues: (i) how to give users the power of controlling and adapting their agents, using a high-level language; and (ii) how to engineer such systems. The first is related to the definition of a high-level meta-model (Section 3), used to build user DSMs to be manipulated by end-users. The later consists of a software architecture that uses the proposed meta-model. In this section, we first present a motivation to our solution, then we detail the architecture.

2.1 Software Engineering Practices to Develop User Agents

One essential characteristic of personalized user agents is that they must store information specific to each user, i.e. configurations and preferences. This is typically realized through code in two different ways: (i) user model: user information is stored in a single location and the model is checked whenever a user-dependent action is performed; and (ii) control variables: variables are inserted in the code, thus reflecting user customizations and these are used to make some decisions that indicate the right course of actions of an agent. Both solutions are essentially the same, with the difference that the first solution modularizes all the user-specific data. Even though these solutions produce the desired behavior, they have some drawbacks, mainly related to the engineering of such systems.

Concentrating all user customizations in a single component creates a high coupling between this component and other components of the system. In addition, changes in this unique component may imply a lot of little changes applied to a lot of different classes. This characterizes the Shotgun Surgery bad code smell [7]. Moreover, in both solutions, a control variable will be used – in (i), it is retrieve from the user model – which is a program variable that is used to regulate the flow of control of the program. These control variables, i.e. user customizations, may be used in several locations of the system and are usually used in chained `ifs` or `switch` statements scattered throughout the system. If a new clause is added to the switch, all statements must be changed. This is another bad code smell, the Switch Statement [7], and the object-oriented notion of polymorphism gives you an elegant way to deal with this problem.

Other software engineering issue related to user agents is that 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 1(a)) or modularizing user settings in a single model (Figure 1(b)). It can be seen that in both approaches it is not possible to modularize concerns in single modules. In addition, without modularizing user customizations, as in Figure 1(a), they are buried into code thus making it difficult to understand them as a whole.

Based on the above arguments, we claim that there is a need for better software architectures to build personalized user agents, which take into account good software engineering practices.

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.

However, dealing with variable traits that emerge from user customization points is not a trivial task. These customization points are spread all over the system architecture and play different roles into agent architectures [6, 12]. We illustrate examples of different roles that user preferences play into agent architectures in Table 1. And 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).

2.2 Detailing our Software Architecture

Our solution to the previously described issues is to provide a *virtual separation of concern* [8]. The main idea is to structure the user agent architecture in terms of services by modularizing its variability as much as possible into agent abstractions. In addition, we provide this virtual modularized view of user customizations, as Figure 1(c) illustrates. Preferences and customizations are not design 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, and users are able to configure and adapt their agents by means of this model. In this section we detail our proposed architecture, depicted in Figure 2, and describe the mechanism that makes the virtual user model (referred to as user model, from now on) work with agent architectures.

The *User Agents* module consists of agents that provide different services for users, such as news, scheduling and trip planning. Their architecture supports variability related to different users, as well as provide mechanisms to reason about preferences. These 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. User agents may share information with other user agents, but the *Security* module aggregates policies that restrict this communication, assuring that confidential information is kept safety se-

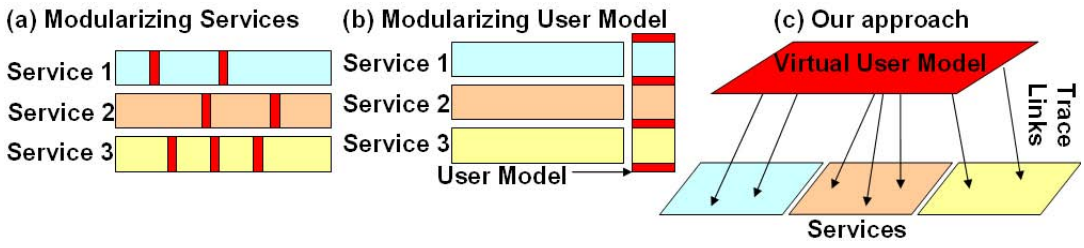

Figure 1: Modularization Approaches.

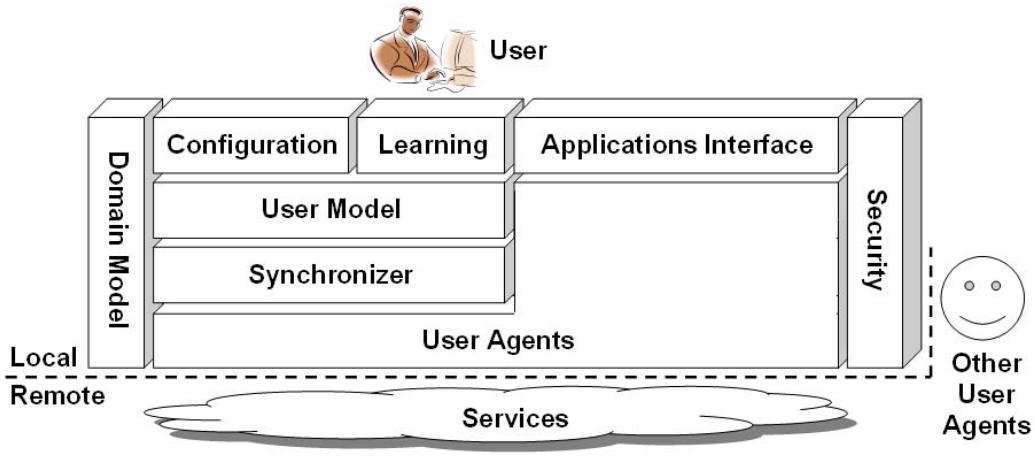

Figure 2: Proposed Architecture.

cured. Services provided to users by user agents have an interface to be accessed (*Applications Interface* module).

As previously discussed, the *User Model* contains user configurations and preferences expressed in a high-level language. They are present in user agents architecture but as design-level abstractions. Clearly, there is a connection from the *User Model* and *User Agents*. This connection is stored in the form of trace links, indicating how and where a configuration or preference is implemented in a user agent(s). For instance, if a `Buyer` agent provides two ways of paying for a product: (a) credit card, and (ii) pay upon pick up. So, there is a variability in the *User Model*, in which the user may choose the payment type. These payment types are implemented as two different plans in the `Buyer` agent that may achieve the `Pay` goal. If the *User Model* has a strong preference stating that the user does not want to pay upon pick up, the `PayUponPickUp` plan is suppressed from the `Buyer` agent architecture, and it will not be considered to achieve the `Pay` goal. In case the *User Model* has a soft preference stating that the user prefers to pay with credit card, both plans will be part of the agent plan library, but a belief containing this preference is added to the agent knowledge base.

These adaptations are performed at runtime and are accomplished based on the trace links between the *User Model* and the *User Agents* architecture. The module that is in charge of adapting *User Agents* based on changes in the *User Model* is the *Synchronizer* module. It is able to understand these trace links, and to know 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*. By means of the *Configuration* module, users are able to 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.

3 A Meta-model to Building Application-specific User Models

In this section, we present and detail our proposed meta-model, whose aim is to allow building application-specific user models, using domain specific abstractions. The meta-model provides concepts to represent user configurations and preferences. Our meta-model, which is an extension of the UML meta-model[1], is depicted in Figures 3 and 4. Elements of the UML meta-model, e.g. `Class`, `Property` and `EnumerationLiteral`, are either distinguished with a gray color in diagrams or are referred in properties.

Before instantiating the meta-model to model user customizations at runtime, it is necessary to build the Domain Model (Section 2) at development time, for defining domain abstractions that are referred to in the User Model. The Domain Model consists of: (i) an Ontology model; (ii) a Variability model; and (iii) a Preferences Definition model. The Ontology model represents the set of concepts within the domain and the relationships between those concepts. The Variability model, in turn, allows modeling variable traits within the domain, which are used later for defining user configurations. In this model we adopted the notation proposed in [14]. The goal of the Variability model is to describe variation points and variants in the system, which can be either optional or alternative. In addition, restrictions may be defined in order to represent relationships between variations. For instance, a certain variant is mutually exclusive to another one. The Variability model is used to define the configuration of the system in the User Model. This part of our meta-model was explored in our previous work [13]. Therefore, we give this brief introduction to the Variability model, but we refer the reader to [14] and [13] for further details.

The part of our meta-model that is used in the Preferences Definition model is presented in Figure 3. The purpose of this model is to define how users can express their preferences and about which elements of the Domain model. Even thought it is desirable that users are able to express preferences in different ways, it is necessary to have agents that are able to deal with them. For instance, if application agents are able to deal only with quantitative preference statements, user preferences expressed in a qualitative way will be meaningless.

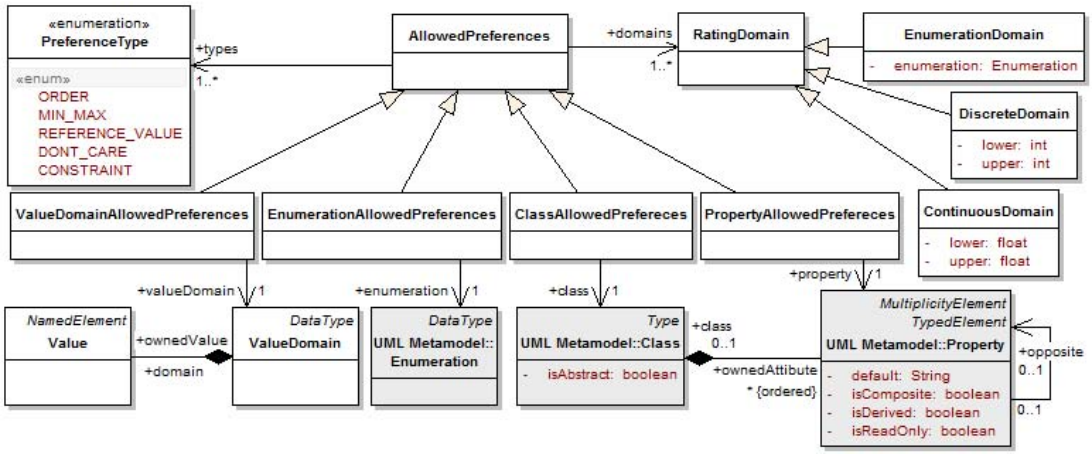

Figure 3: A Meta-model for Modeling User Preferences (Part I).

Users can express different preference types: (i) Order (`PreferenceType.ORDER`) – it expresses an order relation between two elements, allowing the user to express *"I prefer trains*

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

to airplanes." A set of instances of the Order preferences comprises a partial order; (ii) Reference Value (`PreferenceType.REFERENCE_VALUE`) – users are able to indicated a preferred value for an element. This can be interpreted by considering that the user preference is a value on the order of the provided value; (iii) Minimize/Maximaze (`PreferenceType.MIN_MAX`) – this kind of preference indicates that the user preference is to maximize or minimize a certain element; (iv) Don't Care (`PreferenceType.DONT_CARE`) – it allows to indicate a set of elements that the user do not care about. It is useful to users express *"I don't care if I travel with company A, B or C;"* (v) Rating – it allows the user to rate an element. By defining a `RatingDomain` for an element, the user 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. Additionally, an enumeration can be specified, e.g. LOVE, LIKE, INDIFFERENT, DISLIKE and HATE. Different domains can be specified for the same element. Using Rating preferences, it is possible to assign utility values to elements, or express preference statements; and (vi) Constraint (`PreferenceType.CONSTRAINT`) – this is a particular type of preference that establishes a hard-constraint over decisions, as opposed to the other preference types that are used to specify soft-constraints. Constraints allows users to express strong statements, e.g. *"I do not 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 $f(x) = \sqrt[\alpha]{x}$.

Developers must create instances of `AllowedPreferences`, and make the corresponding associations with types and domains, in order to define allowed preference types,. The specializations of `AllowedPreferences` characterizes 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. We adopted this term from [3]. 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 the domain expert that knows what comfort means.

Based on these definitions and our meta-model (Figure 4), 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 variation points from the Variability model, defining their configurations [13]. On the other hand, in the Preferences model, users define preferences and constraints. These are more related to a cognitive model of the user. User preferences (or soft-constraints) determine what the user prefers, and therefore how the system *should* behave, but, if it is not possible, the system may choose other acceptable alternatives. Constraints, in turn, are restrictions (hard-constraints) over elements. As opposed to preferences, they specify certainties in the system.

Figure 4 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. 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 meta-model to make a common superclass of classes, properties, enumerations and values. Given that we did not want to modify the UML meta-model, but only 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. In addition, it allows to specify nested properties, such us `Flight.arrivalAirport.location.country`.

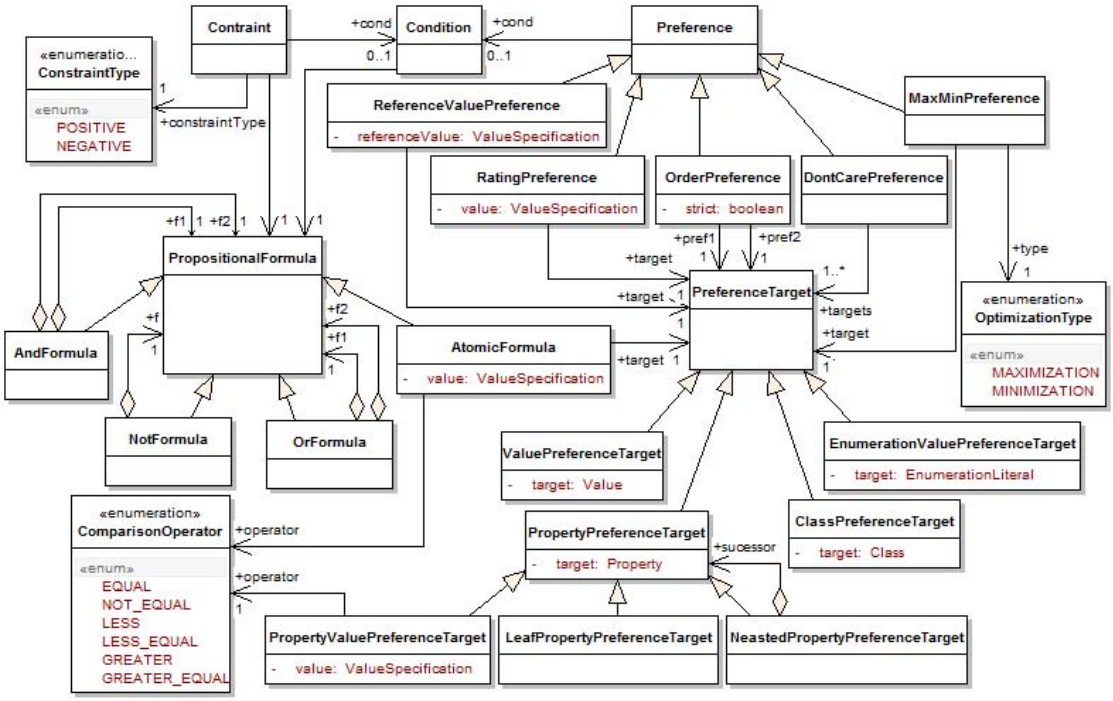

Figure 4: A Meta-model for Modeling User Preferences (Part II).

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

4 Evaluating our User Meta-model across Different Application Domains

Our meta-model was built using preference statements collected from different individuals or from papers related to user preferences. The idea was to contemplate the different kinds of preference statements in order to maximize the users power of expressiveness. In addition, the meta-model uses abstractions used in user preferences domain, therefore the language built based on it is an end-user language. In this section, we present two Preferences models to show that our meta-

model is generic enough to model different kinds of preferences statements in different domains – travel and computer domains. Due to space restrictions, we present only the Preferences model. Nevertheless, given that these are two well-known domains, we assume that the reader is familiar with them. In addition, we assume that all preference types can be expressed in these models.

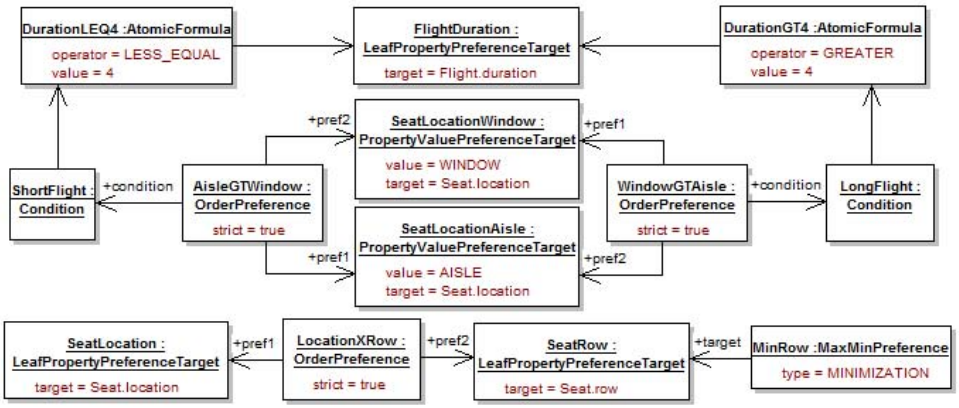

Figure 5: User Preferences model in Travel Domain.

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

P1. *If the flight is short, i.e. its duration does not exceed 4 hours, I prefer to seat in the aisle to seat by the window.*

P2. *If the flight is long, i.e. its duration is higher then 4 hours, I prefer to seat by the window to seat in the aisle.*

P3. *I always prefer to seat in the first rows of the airplane.*

P4. *Seating in the first rows of the airplane is more important to me than the seat location.*

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, which are mobility, readability, performance and cost. In addition, it is specified that 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 for me, I rate it with +++++.*

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.*

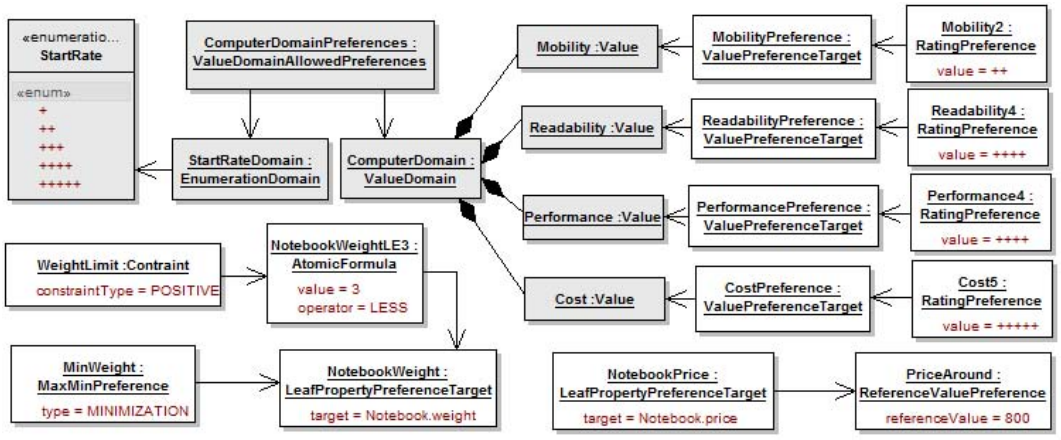

Figure 6: User Preferences model in Computer Domain.

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

5 Related Work

Several approaches have been proposed to deal with user preferences. To build our meta-model, we have made extensive research on which kind 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"). In [1], a framework is proposed, in which a preference function maps records to a score from 0 to 1. On the other hand, qualitative preferences are represented as CP-Nets, as proposed in [5]. Both approaches, qualitative and quantitative, can be represented through our meta-model. CP-Nets also allow to model conditionality, which is considered in our work as well. The concept of normality is defined in [9]. This work argues that users express preferences considering normal states of the world, but these preferences may change when the world changes. We believe the normality abstraction can be modeled using conditions, so we have not adopted it in our model.

Ayres & Furtado proposed the OWLPref [2], 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, our goal is to use a user-oriented approach. Additionally, 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 meta-model is also proposed in [18]. However, its power of expressiveness is very limited. It only allows defining desired values (or intervals) of object properties.

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 their behalf. Agents must have reliable user models that assure they act appropriately, otherwise they will not be trusted by users.

In this paper, we proposed a meta-model that provides abstractions from the user domain, including constraints and preferences. Different abstractions used by end-users in natural language statements are directly represented. Our meta-model provides a domain-specific language that empowers the user to program their agents. Besides constraints, five different preferences types can be represented. In addition, we adopt values as a first-class abstraction to model high-level preferences. Instances of our meta-model are to be used in combination with our proposed architecture, which uses them as a virtual user representation. Services are provided by user agents structured with tradicional agent-based architectures. The User Model provides a modularized view of different user-related concepts spread into agent architectures. A synchronizer module assures that changes in the User Model demands appropriate adaptations in user agents.

We are currently working on a language based on our meta-model using syntactic sugar. In addition, we are investigating how to verify the User Model to identify inconsistencies on preferences.

References

[1] Rakesh Agrawal and Edward L. Wimmers. A framework for expressing and combining preferences. In *2000 ACM SIGMOD*, pages 297–306, 2000.

[2] Leonardo Ayres and Vasco Furtado. Owlpref: Uma representação declarativa de preferências para web semântica. In *XXVII Congresso da SBC*, pages 1411–1419, Brazil, 2007.

[3] Trevor Bench-Capon. Persuasion in practical argument using value-based argumentation frameworks. *Journal of Logic and Computation*, 13(3):429–448, 2003.

[4] Craig Boutilier, Ronen I. Brafman, Carmel Domshlak, Holger H. Hoos, and David Poole. Cp-nets: a tool for representing and reasoning with conditional ceteris paribus preference statements. *J. Artif. Int. Res.*, 21(1):135–191, 2004.

[5] Craig Boutilier, Ronen I. Brafman, Holger H. Hoos, and David Poole. Cp-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *Journal of Artificial Intelligence Research*, 21:135–191, 2004.

[6] Jon Doyle. Prospects for preferences. *Computational Intelligence*, 20:111–136, 2004.

[7] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1 edition, July 1999.

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

[9] Jérôme Lang and Leendert van der Torre. From belief change to preference change. In *ECAI 2008*, pages 351–355, The Netherlands, 2008. IOS Press.

[10] Xudong Luo, Nicholas R. Jennings, and Nigel Shadbolt. Acquiring user tradeoff strategies and preferences for negotiating agents: A default-then-adjust method. *Int. J. Hum.-Comput. Stud.*, 64(4):304–321, 2006.

[11] Pattie Maes. Agents that reduce work and information overload. *Commun. ACM*, 37(7):30–40, 1994.

[12] Ingrid Nunes, Simone Barbosa, and Carlos Lucena. Modeling user preferences into agent architectures: a survey. Technical Report 25/09, PUC-Rio, Brazil, September 2009.

[13] Ingrid Nunes, Carlos J. Lucena, Donald Cowan, and Paulo Alencar. Building service-oriented user agents using a software product line approach. In *ICSR '09*, pages 236–245, 2009.

[14] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[15] Kenneth Rogoff. Grandmasters and global growth. *Project Syndicate*, 2010. Available at http://www.project-syndicate.org/commentary/rogoff64/English.

[16] Ingo Schwab, Alfred Kobsa, and Ivan Koychev. Learning about users from observation. In *AAAI 2000 Spring Symposium: Adaptive User Interface*, pages 241–247, 2000.

[17] Mary Shaw and Paul Clements. The golden age of software architecture. *IEEE Softw.*, 23(2):31–39, 2006.

[18] Dilek Tapucu, Ozgu Can, Okan Bursa, and Murat Osman Unalir. Metamodeling approach to preference management in the semantic web. In *M-PREF 2008*, pages 116–123, USA, 2008.

[19] Gerhard Weiss, editor. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT Press, Cambridge, MA, USA, 1999.