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

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

Pontifical Catholic University of Rio de Janeiro (PUC-Rio) – Rio de Janeiro – Brazil
E-mail: {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 to customize and dynamically adapt their user agents. In addition, we propose a general architecture for developing user-customizable agent-based systems.

1 . Introduction

Many modern computer systems are providing assistance to several of our usual tasks, by incorporating features with a proactive and autonomous behavior. Typical examples include product recommendations based on our purchase history and generation of playlists based on songs we listen. These systems are increasingly becoming part of our everyday life. A generalized and ambitious idea underlying such systems is the personalized user agents [8], which are personal assistants acting on the users' behalf. Even though significant research effort has been invested on developing user agents, we are far from their massive adoption.

Schiaffino & Amandi [11] presented an empirical study that gives a solid basis for explaining this scenario. They claim the "human-computer interaction people have criticized agent-based methodologies that seem to produce systems not easily accepted by the user: one of the main reasons is the autonomy of the agents that can cause a loss of control by the user." Their study showed that different users need different kinds of user agents. In addition, a large group of users is willing to adopt user agents only if they know exactly what the agent is going to do.

Our research addresses this group of users. In this paper we demonstrate an approach to empower users with a high-level domain-specific language that allows them to dynamically program and personalize their agents, as opposed to inference models that might reach the wrong conclusions about user preferences and cause agents to take inappropriate actions. 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. 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 better than others in certain contexts. The present work evolved from our approach for building customized service-oriented user agents [10], which only dealt with user configurations and it did not address dynamic adaptations, i.e. our previous user agents did not evolve at runtime.

Our goal in this paper is twofold. We present a *Domain-specific Model (DSM) to model user preferences*, which provides the necessary vocabulary to build an end-user preferences language. Exiting 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 express different kinds of preference statements, creating a vocabulary that is very close to natural language. The proposed DSM is a metamodel that may be instantiated to build different applications.

We also show how this language is used in broader context, which is an *architecture to build user-customizable applications*, composed of user agents that are dynamically adapted based on a user model that follows our metamodel. We have taken into account software engineering issues identified as current practices to develop applications based on user models. In this sense, we also contribute with an analysis of existing mechanisms to implement user customizations, which may result in low-quality software architectures. Good (modular, stable, ...) architectures are essential to produce higher quality software which is easier to maintain. Otherwise, software architectures may degenerate over time, making their maintenance a hard task, by increasing costs

with refactorings.

This paper is organized as follows. In Section 2, we describe our user-driven software architecture. Section 3 presents our DSM, followed by Section 4, which evaluates our metamodel, showing its generality when used across different domains. Section 5 presents related work. Finally, Section 6 concludes this paper.

2 . A User-driven Software Architecture

Our research on developing personalized user agents is driven by a reference architecture that allows to adapt agents based on an end-user's DSM. In this section, we first present usual software engineering practices adopted to develop user-model based systems. They motivate the structure of our reference architecture, which is also detailed.

2.1 . Software Engineering Practices to Develop User Agents

An essential characteristic of user agents is that they store information specific to each user. This is typically implemented either using: (i) a user model, which stores user information in a single location and is checked whenever a user-dependent action is performed; and (ii) control variables, which are inserted in the code to reflect user customizations and used to make some decisions that indicate to an agent the right course of actions it should take. Both solutions are essentially the same, with the difference that the first solution concentrates all the user-specific data. Even though these solutions produce the desired behavior, they have drawbacks from a software engineering perspective.

Concentrating all user customizations in a single component creates a high coupling between this component and other system components. 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 [6]. Moreover, in both solutions, a control variable will be used – in (i), it is retrieved from the user model – which is a program variable used to regulate the flow of control of the program. These control variables, i.e.

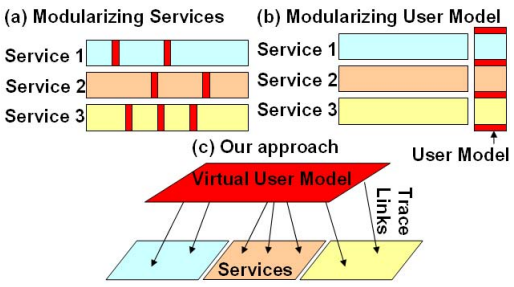


Figure 1. Modularization Approaches.

user customizations, may be used in several system locations and are usually used in chained `if` 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 [6], and the object-oriented notion of polymorphism gives you an elegant way to deal with this problem.

Another 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 it is not possible in either approach to modularize concerns in single modules. In addition, without modularizing user customizations, as in Figure 1(a), they are buried inside the code, thus making it difficult to understand them as a whole.

Based on these arguments, we claim that there is a need for better software architectures to build personalized user agents, taking into account good software engineering practices. 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 in agent architectures [5, 9]. 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 concerns*. 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. We provide a virtual modularized view of user customizations, as Figure 1(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. This section details our proposed architecture, depicted in Figure 2, and describes the mechanism that makes the virtual user model (henceforth referred to as user model) work with agent architectures.

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, as well as provide mechanisms to reason about

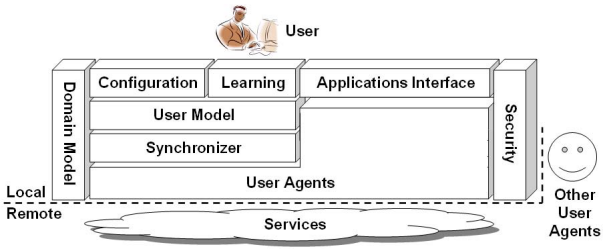


Figure 2. Proposed Architecture.

preferences. These 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 safety 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. 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 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 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*. 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.

3 . A Metamodel for Building Application-specific User Models

In this section, we present and detail our proposed metamodel, whose aim is to allow to build application-specific user models, using domain-specific abstractions. The metamodel provides concepts to represent user configurations and preferences. Our metamodel, which is an extension of the

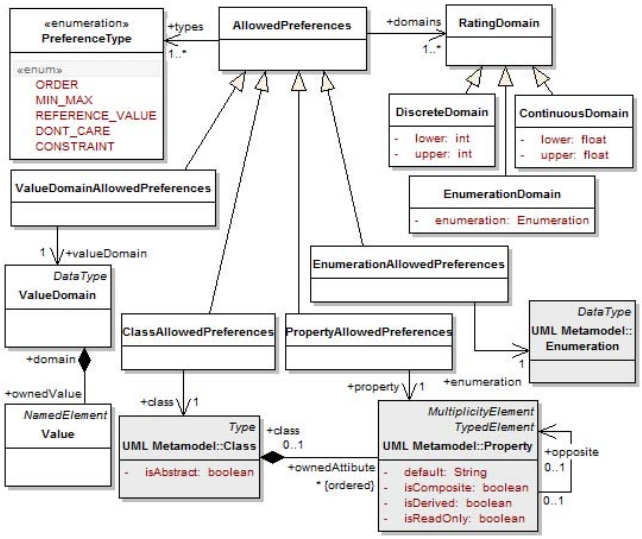


Figure 3. A Metamodel for Modeling User Preferences (Part I).

UML metamodel[1], is depicted in Figures 3 and 4. Elements of the UML metamodel, e.g. `Class` and `Property`, are either distinguished with a gray color in diagrams or are referred in properties.

Before instantiating the metamodel 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 later used for defining user configurations. 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. The Variability model is used to define the configuration of the system in the User Model. This part of our metamodel was explored in our previous work [10] and is out of the scope of this paper. Therefore, we give this brief introduction to the Variability model, but we refer the reader to [10] for further details.

The part of our metamodel 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 though it is desirable that users be able to express preferences

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

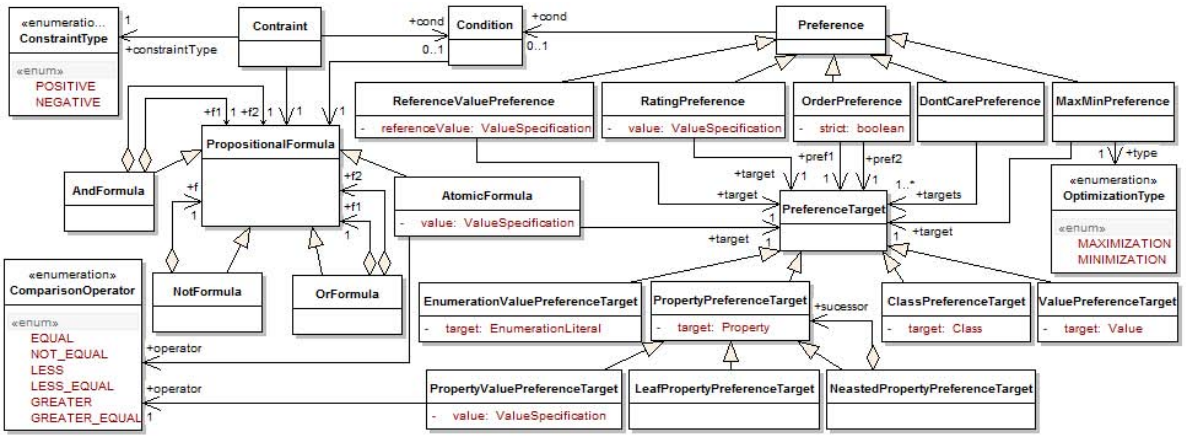


Figure 4. A Metamodel for Modeling User Preferences (Part II).

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.

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`) – allows indicating a set of elements the user does not care about, e.g. *"I don't care if I travel with company A or B;"* (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. 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 a domain expert that knows what comfort means.

Based on these definitions and on our metamodel (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 [10]. On the other hand, in the Preferences model, users define preferences and 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.

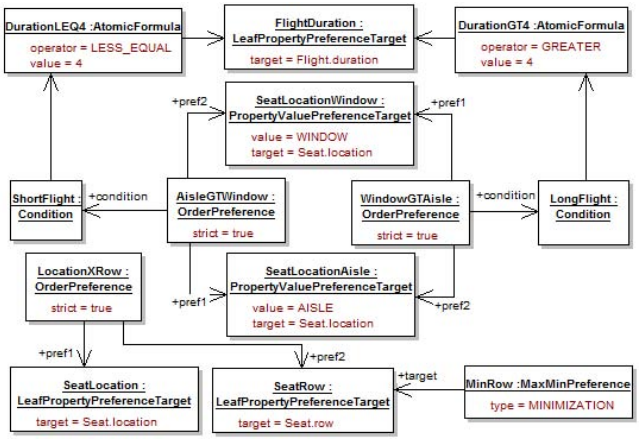
Figure 4 shows the `Constraint` element and five different specializations of `Preference` that represent the different preference types previously introduced. Constraints are expressed in propositional logic formulae, however using only $\neg$, $\wedge$ and $\vee$ logical operators. Atomic formulae 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. 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, 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.

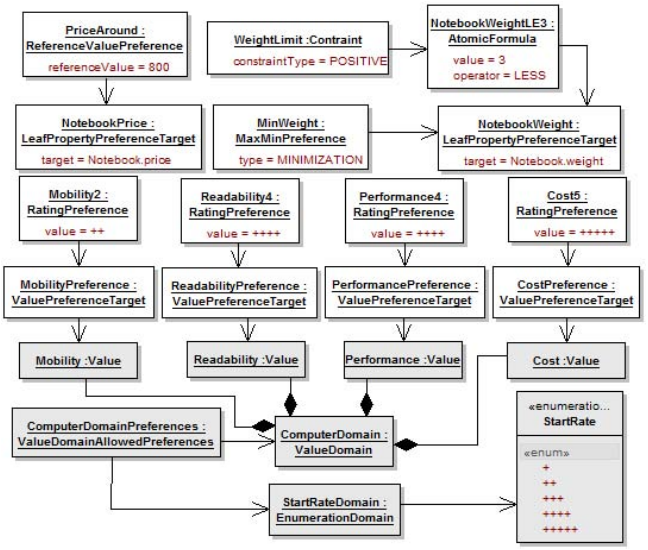
4 . Evaluating our User Metamodel across Different Application Domains

Our metamodel was built using preference statements collected from different individuals 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 and computer 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 seat inside an airplane. This model consists of three order preferences, two



(a) Flight Domain



(b) Computer Domain

Figure 5. User Preferences model.

of them with conditions, and one minimization preference. Next, we present the four modeled preference statements in natural language, and Figure 5(a) 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 then 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 in the first rows of the airplane is more important to me than the seat location.*

The computer domain Preferences model presented in Figure 5(b) 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 5(b) 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 5(b):

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.

5 . Related Work

Several approaches have been proposed to deal with user preferences. To build our metamodel, we have made 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 [1], by means of a preference function that maps records to a score from 0 to 1. On the other hand, CP-Nets [4] models qualitative preferences. CP-Nets also allow modeling conditionality, which is considered in our work as well. The concept of normality is defined in [7], 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 [2], a declarative and domain-independent preference representation in OWL. 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 [12]. However, its expressiveness is very limited. It only allows to define desired values (or intervals) of object properties.

6 . Conclusion

In this paper, we proposed a domain-specific metamodel 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 metamodel provides a domain-specific language that empowers users to express their preferences 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 metamodel 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 traditional agent-based architectures. The User Model provides a modularized view of different user-related concepts spread into agent architectures. A Synchronizer module ensures that changes in the User Model demands appropriate adaptations in user agents.

We are currently working on 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.

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 - G1. Design techniques to improve the development of autonomic Web applications.

References

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

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

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

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

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

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1 edition, July 1999.

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

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

[9] I. Nunes, S. Barbosa, and C. Lucena. Modeling user preferences into agent architectures: a survey. Technical Report 25/09, PUC-Rio, Brazil, September 2009.

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

[11] S. Schiaffino and A. Amandi. User - interface agent interaction: personalization issues. *Int. J. Hum.-Comput. Stud.*, 60(1):129–148, 2004.

[12] D. Tapucu, O. Can, O. Bursa, and M. O. Unalir. Metamodeling approach to preference management in the semantic web. In *M-PREF 2008*, pages 116–123, USA, 2008.