

IST Amigo Project Deliverable D4.7

Intelligent User Services

3 - User Modeling and Profiling Service

Software Developer's Guide

IST-2004-004182

Public

Project Number	:	IST-004182
Project Title	:	Amigo
Deliverable Type	:	Report

Deliverable Number	:	D4.7 (UMPS contribution)
Title of Deliverable	:	3 - UMPS Software developer's guide
Nature of Deliverable	:	Public
Internal Document Number	:	amigo_3_d4.7_final
Contractual Delivery Date	:	30 November 2007
Actual Delivery Date	:	14 January 2008
Contributing WPs	:	WP4
Author(s)	:	Otilia Kocsis (SingularLogic), Elena Vidjiounaite (VTT)

Abstract

This document is the final programmers guide for the User Modeling and Profiling Service components. It describes how to install, use and edit the components of UMPS.

Keyword list

UMPS, User Modeling, Profiling

Table of Contents

Table of Contents.....	2
1 Component Overview	6
1.1 Reasoning Module	6
1.2 Static Modeler	8
1.3 Feedback Analyzer	9
1.4 Context Module	10
1.5 Dynamic Modeler	12
1.6 Multi-Profile Aggregator	14
2 Deployment	16
2.1 Reasoning Module	16
2.1.1 System requirements	16
2.1.2 Download	16
2.1.3 Install	16
2.1.4 Configure.....	16
2.1.5 Compile	16
2.2 Static Modeler	17
2.2.1 System requirements	17
2.2.2 Download	17
2.2.3 Install	17
2.2.4 Configure.....	17
2.2.4.1 Core Service Configuration	17
2.2.4.2 User profile initialization GUI Configuration	17
2.2.4.3 Direct manipulation GUI Configuration	17
2.2.5 Compile	17
2.3 Feedback Analyzer	18
2.3.1 System requirements	18
2.3.2 Download	18
2.3.3 Install	18
2.3.4 Configure.....	18
2.4 Context Module	18
2.4.1 System requirements	18
2.4.2 Download	18
2.4.3 Install	18
2.4.4 Configure.....	18
2.4.5 Compile	18
2.5 Dynamic Modeler	18
2.5.1 System requirements	18
2.5.2 Download	18

2.5.3	Install	19
2.5.4	Configure	19
2.5.4.1	ModelMultimed Service Configuration	19
2.5.5	Compile	19
2.6	Multi-Profile Aggregator	19
2.6.1	System requirements	19
2.6.2	Download	19
2.6.3	Install	19
2.6.4	Configure	19
2.6.4.1	Multi-Profile Aggregation from application log files configuration	19
2.6.5	Compile	19
3	Component Architecture	20
3.1	Service Architectural Model.....	20
3.2	Components Interfaces	21
3.2.1	Reasoning Module.....	21
3.2.1.1	Capability <i>IUserProfileQuerying::getSetting()</i>	21
3.2.1.2	Capability <i>IUserProfileQuerying::getUserProfile()</i>	22
3.2.1.3	Capability <i>IUserProfileQuerying::getUserProfileBranch()</i>	22
3.2.1.4	Capability <i>IUserProfileQuerying::getPreferenceValue()</i>	22
3.2.1.5	Capability <i>IUserProfileQuerying::getBranchState()</i>	23
3.2.1.6	Capability <i>IUserProfileQuerying::getChildClasses()</i>	24
3.2.1.7	Capability <i>IUserProfileQuerying::getClassKeys()</i>	24
3.2.2	Static Modeler	24
3.2.2.1	Capability <i>IUserProfileManagement::addUserProfile()</i>	24
3.2.2.2	Capability <i>IUserProfileManagement::deleteUserProfile()</i>	25
3.2.2.3	Capability <i>IUserProfileManagement::updateUserProfile()</i>	25
3.2.2.4	Capability <i>IUserProfileManagement::setSetting()</i>	25
3.2.2.5	Capability <i>IUserProfileManagement::deleteSetting()</i>	26
3.2.2.6	Capability <i>IUserProfileManagement::setBranchState()</i>	26
3.2.3	Feedback Analyzer	26
3.2.3.1	FeedbackAnalyzer data format	27
3.2.3.2	Capability <i>IUserFeedbackAnalysis::AndOrRule</i>	28
3.2.3.3	Capability <i>IUserFeedbackAnalysis::FindFacts</i>	31
3.2.3.4	Capability <i>IUserFeedbackAnalysis::GetNumberOfFacts</i>	32
3.2.3.5	Capability <i>IUserFeedbackAnalysis::GetPositiveExamples</i>	32
3.2.3.6	Capability <i>IUserFeedbackAnalysis::GetNegativeExamples</i>	32
	Capability <i>IUserFeedbackLogging:: logUserFeedback()</i>	32
3.2.3.7	32
3.2.4	Context Module	33
3.2.4.1	Capability <i>IEventManager:: callBack()</i>	33
3.2.4.2	Capability <i>IRetrieveContext::subscribe()</i>	33
3.2.4.3	Capability <i>IRetrieveContext::unsubscribe()</i>	33
3.2.4.4	Capability <i>IRetrieveContext::query()</i>	33
3.2.5	Dynamic Modeler.....	33
3.2.5.1	Capability <i>IModelMultimed:: GetContextBasedDynamicSuggestionsFromPartOfLogFile ()</i>	34
3.2.5.2	Capability <i>IModelMultimed:: GetContextBasedDynamicSuggestionsFromLogFile ()</i>	36
3.2.5.3	Capability <i>IModelMultimed:: GetNContextBasedDynamicSuggestionsFromPartOfLogFile ()</i>	36

3.2.5.4	Capability IModelMultimed:: GetNContextBasedDynamicSuggestionsFromLogFile ()	36
3.2.5.5	Capability IModelMultimed:: GetFrequencyBasedDynamicSuggestionsFromPartOfLogFile ()	37
3.2.5.6	Capability IModelMultimed:: GetFrequencyBasedDynamicSuggestionsFromLogFile ()	38
3.2.5.7	Capability IModelMultimed:: GetNFrequencyBasedDynamicSuggestionsFromPartOfLogFile ()	38
3.2.5.8	Capability IModelMultimed:: GetNFrequencyBasedDynamicSuggestionsFromLogFile ()	38
3.2.5.9	Capability IModelMultimed:: GetNLastItemsFromLogFile ()	38
3.2.6	Multi-Profile Aggregator	39
3.2.6.1	Capability IMultiProfileAggregation:: GetMultiProfileContextBasedDynamicSuggestionsFromPartOfLogFile ()	40
3.2.6.2	Capability IMultiProfileAggregation:: GetMultiProfileContextBasedDynamicSuggestionsFromLogFile ()	42
3.2.6.3	Capability IMultiProfileAggregation:: GetMultiProfileNContextBasedDynamicSuggestionsFromPartOfLogFile ()	42
3.2.6.4	Capability IMultiProfileAggregation:: GetMultiProfileNContextBasedDynamicSuggestionsFromLogFile ()	42
3.2.6.5	Capability IMultiProfileAggregation:: GetMultiProfileFrequencyBasedDynamicSuggestionsFromPartOfLogFile ()	42
3.2.6.6	Capability IMultiProfileAggregation:: GetMultiProfileFrequencyBasedDynamicSuggestionsFromLogFile ()	43
3.2.6.7	Capability IMultiProfileAggregation:: GetMultiProfileNFrequencyBasedDynamicSuggestionsFromPartOfLogFile ()	44
3.2.6.8	Capability IMultiProfileAggregation:: GetMultiProfileNFrequencyBasedDynamicSuggestionsFromLogFile ()	44
3.2.6.9	Capability IMultiProfileAggregation:: GetMultiProfileNLastItemsFromLogFile ()	44
3.2.6.10	Capability IMultiProfileAggregation:: getMultiProfileUnionOfStaticPrefs ()	45
3.2.6.11	Capability IMultiProfileAggregation:: getMultiProfileMinOfStaticPrefs ()	46
3.2.6.12	Capability IMultiProfileAggregation:: getMultiProfileWeightedAverageOfStaticPrefs ()	46
3.2.6.13	Capability IMultiProfileAggregation:: getMultiProfileWeightedAverageWithoutMiseryOfStaticPrefs ()	47
3.2.7	Functionality used by both DynamicModeler and Multi-Profile Aggregator	47
3.2.7.1	Capability IModelMultimed:: trainSVM(...)	48
3.2.7.2	Capability IModelMultimed:: getSVMrank(...)	49
3.2.7.3	Capability IModelMultimed:: startLearning(...)	50
3.2.7.4	Capability IModelMultimed:: getLearnedRanks(...)	53
4	Tutorial	54
4.1	Install, configure and test functionality of compiled service components	54
4.2	Stereotypes Management	56
4.3	Service Interfaces – how to be used by client applications	58
4.3.1	Use of static modeling functionality - getSetting(...), setSetting(...) methods	58
4.3.2	Use of dynamic modeling functionality - getContextBasedDynamicSuggestions(...), getFrequencyBasedDynamicSuggestions(...) and getNLastItems(...) methods	59
4.3.3	Use of Multi-Profile aggregation functionality	60
4.4	Use of SVM	61
4.5	Use of FeedbackAnalyzer and Context-Dependent Learning functionality	62
4.6	GUI for new profile generation	62
4.7	Direct Manipulation GUI tutorial	66
5	Appendix	74
5.1	User profile ontology	74

5.2	Stereotypes	74
5.3	User profile.....	75
5.4	History Data format for Dynamic Modeling.....	76
5.5	Dynamic Modeling tests with real life TV viewing data	76
5.5.1	Data and pre-processing	76
5.5.2	Experimental protocol and results	77
5.5.3	Conclusions	78
5.6	FAQ	78

1 Component Overview

User modeling and profiling service provides the methodology to enhance the effectiveness and usability of services and interfaces in order to (a) tailor information presentation to user and context, (b) reason about user's future behavior, (c) help the user to find relevant information, (d) adapt interface features to the user and the context in which it is used, (e) indicate interface features and information presentation features for their adaptation to a multi-user environment. These goals are achieved by constructing, maintaining and exploiting user models and profiles, which are explicit representations of individual users preferences. User profile update will be performed using static and dynamic modeling methodologies.

The User Modeling and Profiling Service consists of the following main components, some representing services by themselves:

Reasoning Module. This service is responsible for exploring the user profile and responding to other services requests either for parts of the user profile (collection of user preferences for a particular situation) or discrete preferences. Services and applications can use the *IUserProfileQuerying* interface to explore the profiles.

Static Modeler. This service is responsible for the creation, removal and modification of user profiles at user's or application's request. For the user, a GUI is provided, allowing him to modify preference values and to specify their dependency on context, or to enable or disable the modeling of a series of preferences corresponding to a branch in the user profile. For services and applications the *IUserProfileManagement* interface is provided.

Feedback Analyzer. This service enables reasoning regarding users' implicit feedback, as well as triggering of dynamic modeling whenever implicit or explicit user's feedback indicate that re-training of dynamic models is necessary. The interface provided to log user feedback is the *IUserFeedbackLogging* interface. The interface provided for analysis of implicit user's feedback is *IUserFeedbackAnalysis* interface.

Context Module. This service provides the access to the context history data gathered by the Context Management Service (CMS) based on synchronous queries and asynchronous event-based subscriptions. In order to receive the «context changed» events, the *IEventMonitor* interface is provided.

Dynamic Modeler. This component is responsible for learning user preferences at runtime, using the logged user feedback, application log files and data from the Context Interpreter component of the Context Management service. It is composed by 4 subcomponents: (1) Stereotype Activator, (2) Multimedia Dynamic Modeler, (3) Speech Dynamic Modeler and (4) Automatic Updater. Each of these subcomponents represents services that will be used by the other components of UMPS, and provide the corresponding interfaces, *ITrigger*, *IModelMultimed*, *IModelSpeech* and *IUpdate*.

Multi-Profile Aggregator. This component provides an aggregated profile in case of multiple users found in the same context (i.e. the same room).

1.1 Reasoning Module

Provider

SingularLogic

Introduction

This component is responsible for exploring the user profile and responding to other services requests either for parts of the user profile (collection of user preferences for a particular situation) or discrete preferences.

Development status

Second prototype for .NET available.

Intended audience

Project partners

License

LGPL

Language

C#

Environment (set-up) info needed if you want to run this sw (service)

Software:

- Windows 2K/XP
- Amigo .NET 2.0 Based Programming & Deployment Framework from WP3

Platform

Microsoft .NET Framework v2.0

Tools

Microsoft Visual Studio 2005

Files

The following location in gforge repository contain the source code for the pre-release version:
[amigo]/ius/user_modeling/reasoning_module

Documents

UMPS architecture is described in D4.1 and D4.2. Software developer's guide and installation guide are provided in this document.

Tasks

Refine functionality after users' tests.

Bugs**Patches**

N/A

1.2 Static Modeler**Provider**

SingularLogic, VTT

Introduction

This component is responsible for the creation, removal and modification of user profiles at user's or application's request. For the user, a GUI is provided, allowing him to modify preference values, or to enable or disable the modeling of a series of preferences corresponding to a branch in the user profile.

Development status

Second Prototype for .NET available.

Intended audience

Project partners

License

LGPL license.

Language

C#

Environment (set-up) info needed if you want to run this sw (service)

Software:

- Windows 2K/XP
- Amigo .NET 2.0 Based Programming & Deployment Framework from WP3

Platform

Microsoft .NET Framework v2.0

Tools

Microsoft Visual Studio 2005

Files

The following location in gforge repository contain the source code for the pre-release version:
[amigo]/ius/user_modeling/static_modeler

Documents

UMPS architecture is described in D4.1 and D4.2. Software developer's guide and installation guide are provided in this document.

Tasks

Refine functionality after users' tests.

Bugs**Patches**

1.3 Feedback Analyzer

Provider

VTT

Introduction

This component will enables reasoning on users' implicit feedback and provide data for dynamic modeling. Using this component application can decide whether there is a need or not to update the profile at runtime. In case of explicit feedback the user answers to well-defined system questions, while for implicit feedback meaningful semantics are automatically extracted by the system from user's actions (such as monitoring behavior when watching TV) or comments.

Development status

.NET version, which calls a C++ (dll) reasoning engine is available.

Intended audience

Project partners

License

The software itself is under LGPL license, but it makes use of proprietary binaries/ libraries, written in C++, for which no source code is provided.

Language

C#/C++

Environment (set-up) info needed if you want to run this sw (service)

Software:

Platform

.NET

Tools

Visual C# 2005 Express

Files

C# source code is available. C++ reasoning engine is provided as dll (DynamicModellingAndFeedbackAnalyzer.dll file).

Documents

UMPS architecture is described in D4.1 and D4.2. Software developer's guide and installation guide are provided in this document.

Tasks

Refine functionality after users' tests.

Bugs**Patches****1.4 Context Module****Provider**

IPSI

Introduction

This service provides the access to the context history data gathered by the Context Management Service (CMS) based on synchronous queries and the asynchronous event-based subscriptions.

Development status

First version for .NET available.

Intended audience

Project partners

License

The software itself is under LGPL license, but it might make use of proprietary binaries/libraries for which no source code is provided.

Language

C#

Environment (set-up) info needed if you want to run this sw (service)

PC, Microsoft .NET Framework v2.0, Amigo .NET programming framework (from WP3) Context Broker and Context History components (from WP4/Context Management Service)

Platform

Microsoft .NET Framework v2.0, Amigo .NET programming framework (from WP3)

Tools

Microsoft Visual Studio 2005

Files

The following location in gforge repository contain the source code for the pre-release version:
[amigo]/ius/user_modeling/context_module/

Documents

The Context Module architecture is described in D4.2. Software developer's guide is this document.

Tasks

Bugs

None yet

Patches

None

1.5 Dynamic Modeler

Provider

VTT, SingularLogic

Introduction

This component is responsible for the modification (update) of the user profile using the logging data, resulted from implicit or explicit user feedback. Applications can activate additional stereotypes, as a result of triggering events, or applications can configure UMPS to learn user preferences using Case-Based Reasoning or neural networks (Support Vector Machines) algorithms.

Development status

The first version of Dynamic Modeler is implemented and provides several options. Since research results have shown that applicability of different user modeling methods are strongly user-dependent (see Appendix), Dynamic Modeler provides application with two options: first, to query different user modeling methods separately and to fully control use of the query results; and second, to configure UMPS for fully automatic user modeling.

If application wants full control, it can get separately an estimate of user preference value for the current context from CBR (Case-Based Reasoning) and SVM (Support Vector Machines) and to use these estimated preference values in any way it needs. In this case application should also command when to re-train SVM, and which positive and negative examples of user choices to use in SVM training and in case-based reasoning.

If application prefers automatic Dynamic Modeling, positive and negative examples of user choices will be automatically created by Feedback Analyzer component, and re-training of SVM will be done also automatically (at night time after new examples have been created). In this case the estimate of user preference value for the current context will be calculated as a combination (weighted sum) of user preference values provided by Static (stereotypes-based and explicitly acquired) user profile, CBR and SVM. Weights of each component value depend on how successfully this component estimated user preference values in the past; thus, it is essential to provide sufficient implicit and explicit user feedback via Feedback Analyzer component. For example, when a new user has just arrived, only static model will be used. When sufficient number of positive and negative examples is collected, also conclusions of SVM and CBR will be taken into account. In our research for some users SVM significantly outperformed CBR, while for other users prediction accuracy of CBR was significantly higher; and for some users static model alone was sufficient. Thus, we weight higher the predictions of user model component which have shown better prediction accuracy in the past.

Both CBR and SVM use applications' log files for learning, assuming that log files contain names of items (e.g., names of TV programs, or names of cooking recipes) and contexts when these items were used. If metadata of these items is additionally provided, this metadata will

be also used in reasoning. For example, if application log file contains only names of TV programs, Dynamic Modeler will be only able to learn that program named “5 o'clock show” is what users prefer to watch every workday. However, if additionally metadata is provided, Dynamic Modeler will be able also to learn that users like genre “talk show” and will suggest other talk shows as well.

Research results on dynamic modeling methods are available (VTT), see Appendix.

Intended audience

Project partners

License

The software itself is under LGPL license, but makes use of proprietary binaries/ libraries for which no source code is provided.

Language

C#/C++

Environment (set-up) info needed if you want to run this sw (service)

Software:

- Windows 2K/XP
- Amigo .NET 2.0 Based Programming & Deployment Framework from WP3

Platform

Microsoft .NET Framework v2.0

Tools

Microsoft Visual Studio 2005

Files

C# source code is available. C++ parts are not available as source code, but are provided as dll (integrated in the DynamicModellingAndFeedbackAnalyzer.dll).

Documents

UMPS architecture is described on D4.2. Separate software installation guide will be provided with the Service.

Tasks

Refine functionality after users' tests.

Bugs

N/A

Patches

N/A

1.6 Multi-Profile Aggregator

Provider

VTT

Introduction

This component provides an aggregated profile in case of multiple users found in the same context (i.e. the same room).

Development status

The component is developed and makes aggregation of users' profiles based on the history of user actions as well as aggregation of users' static profiles. The SoA research states that aggregation of static user profiles is not successful if users' preferences differ significantly, thus, the main emphasis in Amigo was on dynamic learning of multi-user environments' preferences. Test results on real life TV viewing data have confirmed the feasibility of this approach, because users' choices in multi-user environments were significantly different from choices of each single user alone. The history of user actions is assumed to be stored in applications' log files, and functionality is same as that of Dynamic Modeller, since multi-user environment is also considered as context.

Intended audience

Project partners

License

The software itself is under LGPL license, but makes use of proprietary binaries/ libraries for which no source code is provided.

Language

C#/C++

Environment (set-up) info needed if you want to run this sw (service)

Software:

Platform

Microsoft .NET Framework v2.0

Tools

Microsoft Visual Studio 2005

Files

Source code is not available completely. Parts which are not available as source code are provided as dll (DynamicModellingAndFeedbackAnalyzer.dll file).

Documents

UMPS architecture is described on D4.2. Separate software installation guide will be provided with the Service.

Tasks

Refine functionality after users' tests.

Bugs**Patches**

2 Deployment

An operational user modeling and profiling service should have installed at least the Reasoning Module and the Static Modeler, the basic components for management and querying of user profiles. User profiles and the library of stereotypes are stored in an Access database for the first software version. Reasoning Module and Static Modeler are accessing the database through a common .NET module (DBInterface). This minimum installation enables only direct manipulation of user profiles. In order to take advantage of enhanced user modeling and profiling methodology, the rest of the components should also be installed, and application must also consider the methodology to provide data for dynamic modeling. The data should be acquired for long enough periods of time and for a large variety of users, to be quantitatively and qualitatively sufficient for the training of users' models. In addition, application and service providers are provided with a Stereotypes Manager module in order to modify the library of stereotypes.

Common requirements for all components:

Amigo .NET programming framework

Microsoft .NET 2.0 Framework v2.0

UMPS common components: Constants.dll, CustomControls.dll, CustomTreeControls.dll, DataObjects.dll, DBInterface.dll, Functions.dll

Databases and configuration files should be located under "C:/UMPS/". If different location is chosen, then "DBInterface.dll" need to be recompiled.

Except of the above common requirements, some components may have additional requirements, which are mentioned in the following sessions if the case.

Common installation procedure: For the core components of the service, there is available an installation file "UMPS.msi", which creates the necessary paths and install all mandatory files for the service to function properly (config files and dlls). This installation file doesn't include the dynamic modeling and multi-profile aggregation components and necessary files.

2.1 Reasoning Module

2.1.1 System requirements

2.1.2 Download

"ReasoningModule.exe", located at:

[amigo]/ius/user_modeling/reasoning_module/trunk/ReasoningModule/bin/Release

2.1.3 Install

Run the executable.

2.1.4 Configure

2.1.5 Compile

Using Microsoft Visual Studio 2005 you can compile a new version of Reasoning Module

2.2 Static Modeler

2.2.1 System requirements

UMPS: UserProfile.dll

2.2.2 Download

Service main core: "StaticModeler.exe", located at:

[amigo]/ius/user_modeling/static_modeler/trunk/ReasoningModule/bin/Release

User profile initialization GUI interface for new profile generation:

[amigo]/ius/user_modeling/static_modeler/trunk/UserProfile/bin/Release

Direct Manipulation GUI interface for editing user profile data: UMPS_UI.exe, located at:

[amigo]/ius/user_modeling/static_modeler/trunk/UMPS_UI/UMPS_UI/bin/Release

2.2.3 Install

Run the StaticModeler.exe to install the service.

2.2.4 Configure

2.2.4.1 Core Service Configuration

All parameters needed by the service to function properly (i.e. database location) are configured in the Constants .NET module (Constants.dll file).

2.2.4.2 User profile initialization GUI Configuration

2.2.4.3 Direct manipulation GUI Configuration

UMPS functionality is based on the user profile ontology (see section 5.1 of this document), and Amigo context ontology. The paths for the user profile ontology and Amigo context ontology should be provided. For the pre-release version also the path to the user profiles directory should be provided.

All three configuration parameters are read from the file "C:\\Amigo\\UMPS\\config\\config.txt"

In the same file there are two more parameters to configure: upper limit and lower limit of value of user preferences. These values are needed in order to inform the users about the scale of preferences:

```
//user preference value must be within these limits
double low_limit = 0;
double high_limit = 1;
```

2.2.5 Compile

Using Microsoft Visual Studio 2005 you can compile a new version of Static Modeler and corresponding GUIs.

2.3 Feedback Analyzer

2.3.1 System requirements

Windows XP.

2.3.2 Download

DynamicModellingAndFeedbackAnalyzer.dll, located at
[amigo]/ius/user_modeling/DynamicModeler/trunk/DynamicModeler/bin/Debug

2.3.3 Install

Store under C:\WINDOWS\system32 directory.

2.3.4 Configure

No configuration is needed.

2.4 Context Module

2.4.1 System requirements

2.4.2 Download

“ContextModule.exe”, located at (non-public):
[amigo]/ius/user_modeling/context_module/trunk/ContextModule/bin/Debug

2.4.3 Install

Run the executable.

2.4.4 Configure

No configuration is needed.

2.4.5 Compile

Using Microsoft Visual Studio 2005 you can compile a new version of the Context Module

2.5 Dynamic Modeler

2.5.1 System requirements

DynamicModellingAndFeedbackAnalyzer.dll

2.5.2 Download

Dynamic Modelling of multimedia: “UMPS_Model_Multimed.exe”, located at:
[amigo]/ius/user_modeling/dynamic_modeler/trunk/Model_Multimed/bin/Release

2.5.3 Install

Run the executable for installing C# parts and store DynamicModellingAndFeedbackAnalyzer.dll under C:\WINDOWS\system32.

2.5.4 Configure

2.5.4.1 ModelMultimed Service Configuration

All parameters needed by the service to function properly (i.e. full path of application log file) are passed as parameters of functions. The required format of input parameters and log files is described below in Components Interfaces section, and it is also described in Program.cs file

2.5.5 Compile

Using Microsoft Visual Studio 2005 you can compile a new version of Dynamic Modeler.

2.6 Multi-Profile Aggregator

2.6.1 System requirements

UMPS: Multi_Profile_Aggregator.dll

2.6.2 Download

Multi-Profile Aggregation from application log files: "UMPS_Multi_Profile.exe", located at:

[amigo]/ius/user_modeling/multi-profile_aggregator/trunk/bin/Release

2.6.3 Install

Run the executable for installing C# parts and store DynamicModellingAndFeedbackAnalyzer.dll under C:\WINDOWS\system32.

2.6.4 Configure

2.6.4.1 Multi-Profile Aggregation from application log files configuration

All parameters needed by the service to function properly (i.e. full path of application log file) are passed as parameters of functions. The required format of input parameters and log files is described below in Components Interfaces section, and it is also described in Program.cs file

2.6.5 Compile

Using Microsoft Visual Studio 2005 you can compile a new version of Multi-Profile Aggregator

3 Component Architecture

3.1 Service Architectural Model

The UMPS has an add-on modular architecture (figure 3.1), as described in deliverables D4.1 and D4.2. The basic inner shell of the architecture is the *Core Profile Service* (CPS). This service handles the profile information storage, the request-response operations to other services and applications layer, the information flow to the middleware, as well as the security issues. At the CPS level, the update of the profile will be based on static modeling methodology. The major components of the CPS, providing the functionality necessary for the interfaces offered to other system services and to the applications domain, are: (a) the *Reasoning Module*, (b) the *Static Modeler*, (c) the *Feedback Analyzer* and (d) the *Context Module*. The *Expanded Profile Service* (EPS), the outer shell of the UMPS, will implement enhanced functionalities of the service, including the *Multi-Profile Aggregator* and the *Dynamic Modeler*.

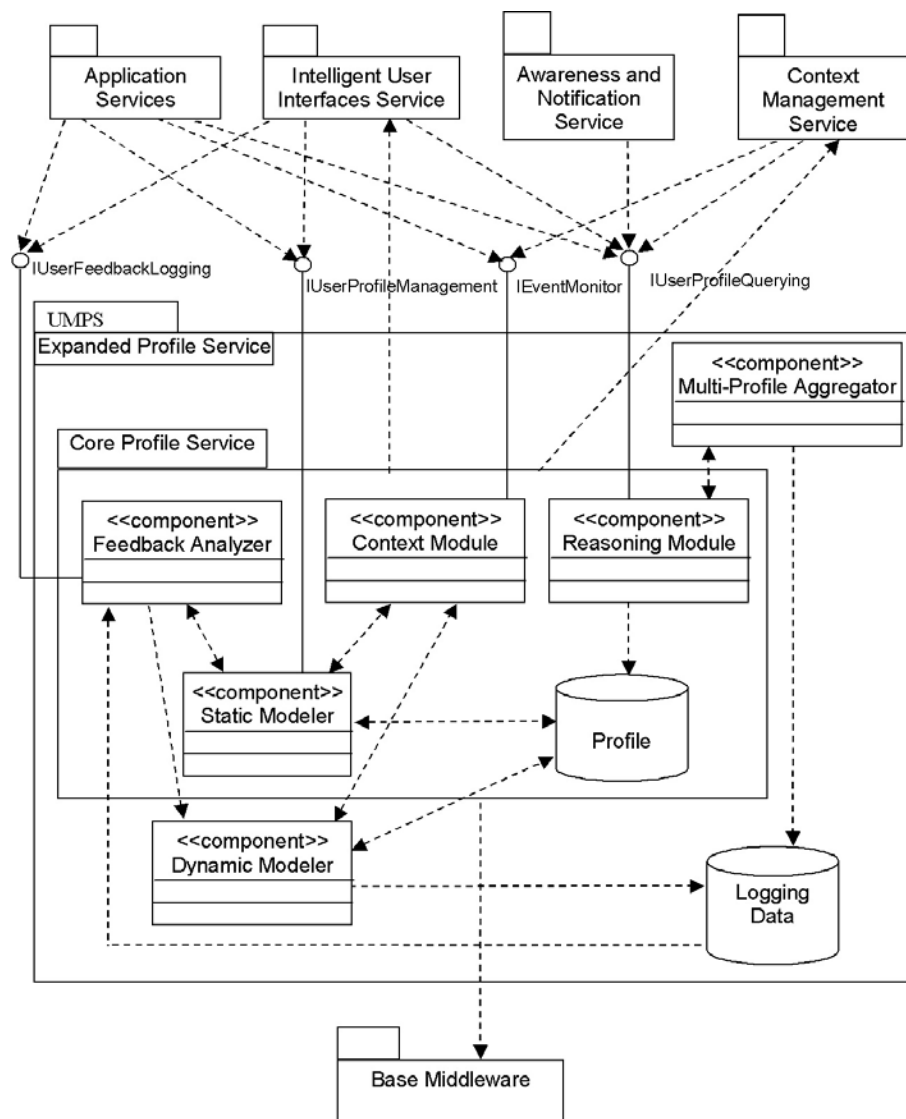


Figure 3.1: Architectural model of UMPS

3.2 Components Interfaces

3.2.1 Reasoning Module

This component offers to other services and application is the *IUserProfileQuerying* interface, to explore user profiles. All capabilities offered through this interface are described in the following section, including detailed information for the implemented ones.

3.2.1.1 Capability *IUserProfileQuerying::getSetting()*

As provided by DBInterface, this capability is a polymorph function, with the following variations

- string getSetting(string userID, string settingID) – returns only the value that has not any parameter
- string getSettingByContext(string userID, string settingID, string contextDescription) – returns the value set for which the parameter “context” has the value given by “contextDescription”
- string getSettingByParameter(string userID, string settingID, string parameterID, string parameterValue) – returns the value set for which the parameter “parameterID” has the value “parameterValue”
- string getSettingAll(string userID, string settingID) – returns all value sets of a key

Description:

- finds the profile of the user identified by userID (unique identifiers in the system)
- finds the setting given by settingID inside the profile, formatted as *"Preferences:MultiMediaPrefs:MoviesPrefs:MovieGenrePrefs:War"*, where the last string is the name of the Key in the profile tree, while the previous ones are classes of preferences.
- if key not found returns "NONE"
- checks the state of all classes before reaching the key, and if any is OFF then the function returns "NONE"
- else returns the corresponding value set(s)

Example function call 1:

```
getSettingAll("DEMO_USER", "Preferences:MultiMediaPrefs:MoviesPrefs:MovieGenrePrefs:War")
```

Returns:

```
<key id="War">
<valueset>
    <value>1</value>
    <rating>100</rating>
    <justification>ANY_PERSON</justification>
</valueset>
<valueset>
    <value>4</value>
    <rating>500</rating>
    <justification>IMPLICIT</justification>
    <parameter id="context">
        <value>DEMO_USER:Company:ALONE</value>
    </parameter>
```

```
</valueset>
</key>
```

Example function call 2:

```
getSetting("DEMO_USER", "Preferences:MultiMediaPrefs:MoviesPrefs:MovieGenrePrefs:War")
```

Returns:

```
<key id="War">
<valueset>
    <value>1</value>
    <rating>100</rating>
    <justification>ANY_PERSON</justification>
</valueset>
</key>
```

Example function call 3:

```
getSettingByContext("DEMO_USER",
"Preferences:MultiMediaPrefs:MoviesPrefs:MovieGenrePrefs:War", "DEMO_USER:Company:ALONE")
```

Returns:

```
<key id="War">
<valueset>
    <value>4</value>
    <rating>500</rating>
    <justification>IMPLICIT</justification>
    <parameter id="context">
        <value>DEMO_USER:Company:ALONE</value>
    </parameter>
</valueset>
</key>
```

3.2.1.2 **Capability IUserProfileQuerying::getUserProfile()**

```
bool getUserProfile(string userID)
```

Description: it is used to retrieve the entire user profile, which is saved in an xml file in the "../UMPS/UserProfiles/" directory. The file name is given by the userID parameter.

3.2.1.3 **Capability IUserProfileQuerying::getUserProfileBranch()**

```
string getUserProfileBranch(string userID, string branchID)
```

Description: it is used to retrieve a profile branch data, in xml format. The method will return this data as an xml formatted string.

3.2.1.4 **Capability IUserProfileQuerying::getPreferenceValue()**

```
string getPreferenceValue (string [] userIDs, string [] metadata, string domain, bool allInOneString)
```

Description: it is used to calculate user preference value for a set of metadata descriptors in one call. For example, a movie recommender application can submit a list of metadata descriptors of a movie (such as genre, names of actors, director and so on) and a list of users

who want to watch movie, and to get back a rank of the movie (instead of querying for preferences of each user for each metadata descriptor). Preferences of each user are calculated as weighted average of preference values for each metadata descriptor, where weights are also taken from the user profile if the user has set them, otherwise default (fairly small) weight values are used. For movies domain, weights of user preferences are taken from profile brunch "relativeImportanceOfMovieMetadata", which embraces such settings as "relative importance of movie actors", "relative importance of movie genre", "relative importance of movie age", "relative importance of generic interests" and so on. That is, if some user mainly pays attention to movie actors when he/ she selects movies to watch, for this user "relative importance of movie actors" setting should be set high. If another user prefers new movies, "relative importance of movie genre" should be set high.

"Relative importance of generic interests" setting is applied to a topic of a movie or TV program (what are they about). For example, if some person is generally interested in dogs (or architecture, or medicine), this person might be also interested in movies or TV programs on these topics. Then "Relative importance of generic interests in movie domain" setting should be set high for such person. On the other hand, a user might prefer to read books or web pages on these topics instead of TV programs. Then "Relative importance of generic interests in books domain" setting should be set high, and in movie domain – low.

Since users can also make own profile more detailed by adding new terms, they can set different settings for e.g. relative importance of generic interest in dogs and generic interest in architecture, and this way to control that TV programs about architecture will be recommended to them, while TV programs about dogs not (or vice versa).

String "domain" parameter controls which profile brunch to use in reasoning.

Bool "allInString" parameter allows to differentiate between two metadata formats which the function can handle. If this parameter is **true**, the function will assume that all metadata about one movie is all included in one owl file provided by Amazon service (work of WP3 and WP6, for details please see the description of how Media Manager Core retrieves metadata and in which format it stores it), and the function will use ContentWorld ontology (same reference) for finding correspondence between user preferences and metadata format. In this case the function can calculate ranks of several movies (as many as string [] metadata size)

If "allInOneString" parameter is **false**, the function assumes that all metadata in the array belongs to one movie and is in a format which directly corresponds to UMPS ontology. In this case each item in metadata array should be as following: "MovieGenre: comedy" is one array item, "CountryOfOrigin: USA" another array item and so on (order does not matter).

This function returns ranks of items in xml format:

```
<key id="Matrix"><valueset><value>-2</value></valueset></key>
```

The ranks range depends on UMPS scale. Since UMPS allows users to set values between -5 and 5, the movie ranks will be also within these limits.

3.2.1.5 Capability IUserProfileQuerying::getBranchState()

string getBranchState(string userID, string branchID)

Description: allow retrieval of the modeling state of a profile branch. The returned values are "ON" || "OFF", meaning that the user has either enabled or disabled the modeling of a profile branch. If the state is "OFF" the profile data corresponding to that branch cannot be used by any application for personalization or adaptation to user preferences (UMPS will not return the values for those settings when getSetting() method is called).

3.2.1.6 Capability IUserProfileQuerying::getChildClasses()

```
string [] getChildClasses(string userID, string parentID)
```

```
string [] getChildClassesFromOntology(string parentID)
```

3.2.1.7 Capability IUserProfileQuerying::getClassKeys()

```
string [] getClassKeys(string userID, string parentID)
```

```
string [] getClassKeysFromOntology(string parentID)
```

3.2.2 Static Modeler

This component offers to other services and application the *IUserProfileManagement* interface for direct manipulation of user profile data.

3.2.2.1 Capability IUserProfileManagement::addUserProfile()

Function:

- bool addUserProfile (string userID)

Description: this function is called in order to generate a new user profile, for the user identified in the system by "userID".

When this function is called, a GUI will be displayed to gather some compulsory information about the user, such as the personal details (first name, last name, date of birth, gender, etc.), as well as some additional information with respect to its preferences, relationships with family members and any other information that the user is willing to provide and can help for a better initialization of its profile (i.e. occupation, religion, severe health problems, un-tolerated products etc.). This initial information is analyzed in order to extract triggers that are activating some stereotypes for initial user profile generation. Table 3.1 shows some rules for triggers extraction. This table will be updated in agreement to the library of stereotypes that will be provided with the service.

condition1	condition2	condition3	trigger
			Person
age<=18			Child
age>18			Adult
age<=18	Gender="Male"		Male_Child
age<=18	Gender="Female"		Female_Child
age>18	Gender="Male"		Adult_Male
age>18	Gender="Female"		Adult_Female
age<=18	age>=12		Teenager
Occupation="Working Person"			Working_Person
Occupation="Studying Person"			Studying_Person
Occupation="Working Person"	Profession="..."		<i>Selected Option</i>

Occupation="Studying Person"	School Level="Preschool"		Preschool_Puple
Occupation="Studying Person"	School Level="School"		"School_Puple"
Occupation="Studying Person"	School Level="Highschool"		"Highschool_Puple"
Occupation="Studying Person"	School Level="University"		"Student"
Religion="..."			<i>Selected Option</i>
Disabled Person checked			Disabled_Person
Disability="..."			<i>Selected Option</i>

Table 3.1. Triggers' extraction rules for initialization of new user profiles.

Whenever a new profile is generated, the first trigger is "Person", activating the ANY_PERSON stereotype.

3.2.2.2 **Capability *IUserProfileManagement::deleteUserProfile()***

Used to remove an user profile from the database.

3.2.2.3 **Capability *IUserProfileManagement::updateUserProfile()***

Information will be provided in the next versions, according to schedule.

3.2.2.4 **Capability *IUserProfileManagement::setSetting()***

This capability is a polymorph function, with the following variations

- bool setSetting(string userID, string settingID, string valueType, string settingValue)
- bool setSettingByContext(string userID, string settingID, string valueType, string settingValue, string contextDescription)
- bool setSettingByParameter(string userID, string settingID, string valueType, string settingValue, string parameterID, string parameterValue)
- bool setSettingMultipleValues(sting userID, string settingID, string valueType, string [] settingValues)
- bool setSettingByContextMultipleValues(string userID, string settingID, string valueType, string [] settingValues, string contextDescription)
- bool setSettingByParameterMultipleValues(string userID, string settingID, string valueType, string [] settingValues, string parameterID, string parameterValue)

Description:

- finds the profile of the user identified by userID (unique identifiers in the system)

- finds the setting given by settingID inside the profile, formatted as "Preferences:MultiMediaPrefs:MoviesPrefs:MovieGenrePrefs:War", where the last string represents the keyID
- checks the state of all classes before reaching the key, and if any is OFF then the function returns "FALSE"
- if any class doesn't exist it will be added, with state ON
- if no matching value set is found, then a new valueset is added to the key
- return "TRUE" if setting modified successfully

3.2.2.5 Capability IUserProfileManagement::deleteSetting()

This capability is a polymorph function, with the following variations

- bool deleteSetting(string userID, string settingID)
- bool deleteSetting(string userID, string settingID, string contextDescription)
- bool deleteSetting(string userID, string settingID, string parameterID, string parameterValue)

Description: it is used to delete preference values (valuesets) of a key, described by the settingID input value, of a certain user profile (identified by the userID input parameter).

3.2.2.6 Capability IUserProfileManagement::setBranchState()

bool setBranchState(string userID, string branchID, string state)

Description: used to set the modeling state of a profile branch. Should be used only for explicit setting of the modeling state by the user, and this decision should not be taken by any application or service. For the state, two values are allowed, "ON" || "OFF".

3.2.3 Feedback Analyzer

Through this component applications and services can update the profile at runtime, based on the feedback from user-system interactions.

This component provides a set of functions, which can be used by applications for reasoning about user feedback in application-specific way, and it also provides functions which are used by Amigo multimedia recommender application for automatic dynamic modeling. Feedback Analyzer combines context data and logs of TV/ movie player for automatic dynamic modeling, and it uses also explicit user feedback data (which applications can provide via logFeedback interface) for this purpose.

Since implicit user feedback can be derived by analyzing application log files, context histories and interactions with user interface service, we developed a reasoning engine for this purpose. Reasoning engine operates on data collected by ContextInterpreter component of Context Management Service; thus, it requires that applications and user interface service provide ContextSource interface if their data should be used for evaluation of implicit or explicit user feedback.

For example, TV programs recommender application can estimate implicit user feedback from user actions: if a user switched to another channel soon after starting watching the first recommended program, probably the first program was not the best suggestion for him. In order to detect such feedback, it is needed to find "start watching the first program" and "switch to another channel" events and to check that second event took place soon after the first event. On the other hand, if time interval between "switch" event and "start watching" event

exceeded one hour, such programs are good candidates for serving as positive examples of user tastes, especially if other context sources confirm that the user was indeed present in front of TV and was not sleeping, or that the users had not reduced sound level and were not discussing holiday plans without paying any attention to TV. Since evaluation of implicit users feedback is generally a challenging, heavily domain-dependent task, and since its success depends on capabilities of context sources (for example, whether topic recognition sensor exists at home, and whether it is capable of detecting that users are discussing holiday plans), we provide generic rule-based reasoning functions, and applications can build rules and specify rule actions.

Applications can call functions of FeedbackAnalyzer, and they can also configure FeedbackAnalyzer to act on their behalf.

First of all we describe how to call functions if applications simply want to receive conclusions regarding user feedback. FeedbackAnalyzer provides the following functions for this purpose:

1. string AndOrRule(string [] leftHandSide, string leftHandSideOperator, string leftHandSideOperatorLast, string filter, string rightHandSide, string whatToReturn)
2. string findFacts (string filter, int minNumberOfFactsToExist, string rightHandSide, string whatToReturn)
3. string getNumberOfFacts (string filter, string rightHandSide)

All these functions have same input parameter, “filter”. This parameter specifies which data from history to use for reasoning. (Details will be discussed below) Another parameter, string whatToReturn, simply allows to restrict reasoning conclusion to a set of context types, specified in whatToReturn string, and corresponding values. For example, to return only “true” or “false” conclusion or also ID of the user who produced “true” conclusion (if conclusion is “false”, nothing else will be returned).

Another common parameter, string rightHandSide, allows to attach additional information to return string. For example, an application can define a rule *“IF 3D gesture “ThumbsUp” was detected within a minute time interval after Speech Interface asked the user “did you like the video”? THEN user explicit feedback is positive”*

“User explicit feedback is positive” is exactly this additional information, which will be attached to the return string, and which is contained in string rightHandSide. (We call it “additional” meaning that it was not present in the data used in reasoning).

3.2.3.1 FeedbackAnalyzer data format

FeedbackAnalyzer uses log files of Context Interpreter component of CMS for reasoning, and it assumes that all facts are sets of type-value pairs. For example, the fact “Jerry made gesture “ThumbsUp” in the living room at 5 p.m.” is expressed as a set of type-value pairs:

```
<type>Source</type><value>GestureUI</value><type>PersonID</type><value>Jerry</value>
<type>Gesture</type><value>ThumbsUp</value><type>TimeStamp
</type><value>2007/12/1 17:00</value><type>Location</type><value>Living room</value>
```

FeedbackAnalyzer assumes that all relevant information is contained in one string (one line of file). Context types are taken from Amigo ontology because Context Sources provide information in such terms, but applications can add facts with own context types as well. A few predefined terms should not be used freely, however, because these terms trigger certain behaviour of FeedbackAnalyzer. (One example of such predefined terms is “Time” term: all time-related descriptors should contain “Time” substring, while other descriptors should not).

Context sets can be of any size, but they can contain each context descriptor only once. For example, the fact that both Jerry and Maria are in the kitchen is represented by the following set of context descriptors:

```
<type>Source</type><value>Positioning</value><type>PersonID</type><value>Jerry⌘
Maria</value><type>Location</type><value>Kitchen</value><type>TimeStamp
</type><value>2007/12/1 17:00</value>
```

None of context descriptors is obligatory to use, although normally each fact has its source and timestamp (time when this fact was created). Other time-related information can be represented by other context descriptors, containing “Time” substring. For example, starting time of the appointment can use “StartTime” type.

3.2.3.2 Capability IUserFeedbackAnalysis::AndOrRule

```
string AndOrRule( string [] leftHandSide, string leftHandSideOperator, string
leftHandSideOperatorLast, string filter, string rightHandSide, string whatToReturn)
```

Description: AND, NotAND and OR rules are traditional reasoning rules. AND rule works as follows: “*IF (fact1 exists AND fact2 exists... AND factN exists) THEN conclusion*”, OR rule checks whether one of the listed facts exists. NotAND rule works as follows: “*IF (fact1 exists AND fact2 does not exist... AND factN does not exist) THEN conclusion*”

Return value: “true” or “false” result, and in case of “true” also **fact1** which produced this result. Whether complete fact1 or parts will be returned depends on whatToReturn parameter, as described above.

Parameter *string leftHandSideOperator* can be OR, AND, NotAND, and these are predefined terms.

Parameter *string [] leftHandSide* is the set of facts (**fact1**, **fact2** ... **factN**). Fact1 can be a set of type-only and type-value pairs, other facts shall be type-value pairs.

Syntax of *leftHandSide* strings determines which parts of stored facts will be compared and how. First, applications can specify all details of **facts** to be compared, e.g., to create a rule

```
leftHandSide [0] =
<type>PersonID</type><value>Jerry</value><type>Location</type><value>Room1</value>
<type>ApplicationQuery</type><value>FeedbackQuery</value><type>Subject</type><value>Matrix
</value>

leftHandSide [1] =
<type>PersonID</type><value>Jerry</value><type>Gesture</type><value>ThumbsUp</value><type>
Location</type><value>Room1</value>

leftHandSideOperator = AND

rightHandSide = <type>ExplicitFeedback</type><value>Positive</value>
```

This rule will only check how Jerry reacted at query about “Matrix” in Room1. However, since the rule had not specified the source of “ThumbsUp” gesture, the rule will check data of all possible sources, e.g, pressing a designated GUI button can also result in appearance of ThumbsUp gesture in log file.

Applications can also create more generic rules, e.g., the following left-hand side strings will request to infer “positive user feedback” conclusion for any user in any room and about any subject:

```
leftHandSide [0] = <type>PersonID</type><type>Location</type>
<type>ApplicationQuery</type><value>FeedbackQuery</value><type>Subject</type>
leftHandSide [1] = <type>Gesture</type><value>ThumbsUp</value>
```

In the first fact of this rule (leftHandSide[0]) there is one type-value pair (ApplicationQuery:FeedbackQuery) and three type-only descriptors: PersonID, Location and Subject

As already stated, adding type-only descriptors to other facts than the first one is not allowed. Type-only descriptors in fact1 work in a following way:

“true” conclusion will be inferred only when values of these types is equal to each other in all facts listed in the rule. If e.g. fact2 contains only some of the listed types, only values of these types will be compared (and should match) values of other facts. So both facts

```
<type>PersonID</type><value>Jerry</value><type>Gesture</type><value>ThumbsUp</value><type>
Location</type><value>Room1</value>
```

and

```
<type>Gesture</type><value>ThumbsUp</value><type> Location</type><value>Room1</value>
```

will produce same result if application query was made in room1. (Only ThumbsUp gesture made in the same room where ApplicationQuery was sent will produce “true” result, because Location type-only descriptor is listed in fact1).

In case of “true” result, application can get back the whole fact1 or its parts. For example, if

```
whatToReturn = <type>PersonID</type><type>Subject</type>,</type>
```

the return result will be

```
<type>ExplicitFeedback</type><value>Positive</value><type>PersonID</type><value> Jerry
</value><type>Subject</type><value>Matrix</value>
```

Parameter *string leftHandSideOperatorLast* can be either OR or AND; these are predefined terms. These parameter controls whether all values of type-only descriptors (listed in fact1) should produce “true” in order the rule produces “true”, or just one value. For example, three family members received a query “did you like the video Matrix?” In case of “explicit positive feedback” rule described above, *leftHandSideOperatorLast* = OR would mean that application is interested whether either one of family members responded positively, while AND would mean that all three persons should respond positively in order the rule produces “true” result.

Now let's discuss parameter **String filter**: it is one of the most important parameters of the rules because it controls which data will be used for reasoning.

Filter is a set of type-value pairs and type-only fields, and filter allows to select (from the stored data) only the data which matches the filter. If value in type-value pair is not numerical value, filter will select for further processing only facts containing type-value pairs which exactly match type-value pairs in the filter. (Type-value and type-only fields of filters are concatenated in AND fashion). For example, filter

`<type>PersonId</type><value>Jerry</type>Location </type> <value>Kitchen</value>`

will leave for further processing only facts about Jerry in the kitchen, independently on what and when Jerry did something in the kitchen. Filter “lightLevel: low” will leave for further processing only facts which contain this particular sensor value in any location of a house and any time.

Unlike facts, filters can have only one value associated with each context descriptor. That is, although

`<type>PersonID</type><value>Jerry& Maria</value>` is a valid fact, it is not a valid filter. It is also important to use type-value pairs which match the data.

For example, filter `<type>PersonID</type><value>Kitchen</value>` will leave no data for further processing because Kitchen is not a person in Amigo data.

Unlike facts, filters can be “type-only” things and to include context type without any value, e.g., filter “lightLevel” will leave for further processing only facts which contain “lightLevel” context descriptor, but its value can be any.

If certain context descriptor is used with numerical values, it is possible to use filter also for selecting facts which are greater than or less than filter value. In this case filter can include additional field “operator”, which is used for “greater than” or “less than” selection.

For example, if timestamps and appointment times are numerical, applications can filter facts within certain time period. For example, filter

`<type>TimeStamp</type><value>2007/12/1 10:55</value><operator>GreaterThan</operator>`

filter will take only facts which are more “fresh” than time specified by a filter.

Filter

`<type>TimeStamp</type><value>2007/12/1 20:55</value><operator>LessThan</operator>`

will take only fairly old facts. Using both filters will result in a data between start and stop timestamps. Similarly, it is possible to select appointments according to their start time and end time if filter types matches with appointments syntax, and appointments times contain a predefined substring “Time”.

Generally, similar to facts, filters can include each context descriptor only once, and numerical “greater than” and “less than” filtering is the only exception: such filter contains same context descriptor twice.

One more time-related filter, “time slot filter”, was developed specially for AND and NotAND rules. It uses predefined term “TimeSlot” and is used when it is important to check that rule facts happened within small time interval. (TimeSlot filter in NotAND rule checks that facts did not happen within small time interval).

For example, home system is such that positioning sensor knows who is in the room, but 3D gesture user interface does not know who performed gesture (quite common situation, indeed). Application asks the user “did you like the video?”, and then user explicit feedback should be considered positive only if ThumbsUp gesture was made soon after the query.

Another example is evaluation of implicit feedback. For example, TV recommender application logs how users switch TV channels, and positioning sensor tracks people. If switch to another channel took place soon after a new person entered TV room, may be this person strongly disliked currently viewed program? Thus, it makes sense to ask the new person whether this was the case, but “ask for explicit user feedback” action should be only triggered if two events

(room entry and channel switch) took place one after each other with a small difference in timestamps.

Similarly, if application wants to deduce implicit user feedback from users' viewing behaviour, it needs to find "start watching the first program" and "switch to another channel" events and to check that second event took place soon after the first event. User models, which learn user preferences from history of user actions, are usually re-trained once a day or once a week, but not right after a new case is added to the history. So for finding new negative examples for re-training application needs to find all cases within last week, when "switch" event happened after "start watching" event within e.g. 10 minutes or within other application-specific time interval. On the other hand, if time interval between "switch" event and "start watching" event exceeded one hour, such programs are good candidates for serving as positive examples of user tastes, especially if other context sources confirm that the user was indeed present in front of TV and was not sleeping.

Syntax of TimeSlot filter is the following:

```
<type>TimeSlot</type><value>0:05:0</value><operator>EventsOrderMatters</operator>
```

<value>0:05:0</value> means that time interval between fact1 and fact2, fact2 and fact3 etc should be less than 5 minutes. Operator "EventsOrderMatters" allows to specify that it is important also to check that fact2 happened after time1, fact3 – after time 2 and so on.

Filter <type>TimeSlot</type><value>0:05:0</value> will not care about order of events, only about time interval between them.

Currently it is not possible to create rules of the kind "if light intensity in Room1 is greater than 5 AND light intensity in Room2 is less than 1 AND Jerry is in Room4". However, application can create two separate rules "if light intensity in Room1 is greater than 5 AND Jerry is in Room4" and "light intensity in Room2 is less than 1 AND Jerry is in Room4" by using a filter "light intensity should be greater than 5" for the first rule and "light intensity should be less than 1" for the second rule.

Time slot filter and related parameters do not have any affect on OR rule.

In case of NotAND rule filter

```
<type>TimeSlot</type><value>0:05:0</value>
```

will produce true is time interval between facts was greater than 5 minutes.

For example, positive implicit user feedback can be detected with this rule: if a user started to watch TV program and did not switch to another channel during an hour, may be he liked this program?

3.2.3.3 Capability IUserFeedbackAnalysis::FindFacts

string findFacts (string filter, int minNumberOfFactsToExist, string rightHandSide, string whatToReturn)

This function returns true and list of found facts (or just one fact, if they were all identical) if number of facts left after using filter exceeded minNumberOfFactsToExist parameter.

For example, if speech user interface recognizes the sentence "Terminator is very stupid video", user feedback about Terminator can be derived just from this fact. If user performed "FastForward" action 10 times during watching a video he never seen before, user feedback about this video is also quite clear.

3.2.3.4 Capability IUserFeedbackAnalysis::GetNumberOfFacts

string getNumberOfFacts (string filter, string rightHandSide)

This function simply returns number of facts left after using filter. For example, if user listened music of Beethoven 100 times during last month, probably the user likes Beethoven.

3.2.3.5 Capability IUserFeedbackAnalysis::GetPositiveExamples

string GetPositiveExamples(string filter, double PositivePercent, int MaxNumOfEntries, string ReturnTerm)

This function was developed for Amigo movie/ TV recommender application, it analyses application log file and context data and calculates viewing percent of a movie (with respect to an overall duration of movie) for each user. Timestamps of user locations are used to calculate how much time each user spent in TV room during playback of a movie or TV program (it is assumed that if the user was in the room during playback of a movie or TV program, the user was watching this movie or TV program). This function also calculates how many times the users entered TV room during playback. An example is considered positive for some user if the user's presence during playback exceeds PositivePercent parameter, and if number of user entries is smaller than MaxNumOfEntries parameter.

If no context data for analysis of implicit user feedback is available, it will be estimated only from viewed percent calculated from application log files, but this way is less intelligent.

3.2.3.6 Capability IUserFeedbackAnalysis::GetNegativeExamples

string GetNegativeExamples(string filter, double NegativePercent, int Min1NumOfEntries, int Min2NumOfEntries, string ReturnTerm)

Similar to a previous function, this function analyses application logs and context data. An example is considered negative if the user was present in TV room less than NegativePercent time, but his number of entries in TV room was greater than Min1NumOfEntries parameter. This is needed because for some applications an example can be considered negative if the user had not even entered a room where activity took place. However, applications can be more certain that an example is negative if the user had started watching TV program or a movie and stopped. Parameter Min2NumOfEntries is used in another way: if number of user's entries to TV room exceeds Min2NumOfEntries, the example is considered negative even if its viewing percent is greater than NegativePercent value.

If no context data for analysis of implicit user feedback is available, it will be estimated only from viewed percent calculated from application log files, but this way is less intelligent.

3.2.3.7 Capability IUserFeedbackLogging::logUserFeedback()

bool logUserFeedback(string logType, string userID, string feedback)

Description:

- different types of feedback can be logged, and depending on the logType, the feedback has the corresponding structure;
- in case of logType="trigger", the feedback should be in the form of *action:TriggerName*, where for the action parameter there are two possible values, "activate" || "deactivate", and the TriggerName should be found between the trigger defined in the stereotypes library.
- in case of logType="implicit" and of logType="explicit", the feedback should contain the data necessary to set a value setting, for which the feedback is provided. In case of feedback

provided for re-training of dynamic models, the feedback should be in a form of context – item name – log file.

- in case of logType="speech", the log data for dynamic speech interface modeling should be provided.

3.2.4 Context Module

The Context Module component offers the external interface *IEventMonitor*, which provides the capability to receive context changed events from the Context Histories component of the Context Management Service. Also, the Context Module component offers the internal interface *IReceiveContext*, providing the capability to subscribe and unsubscribe to the changes of the context history data, as well as to query the context history database. All the capabilities offered through these interfaces are described in the following section, including detailed information for the implemented ones.

3.2.4.1 Capability *IEventMonitor::callBack()*

void callBack(string contextChangedEvent, int SubscriptionID) – the arguments of this method might change, depending on the .NET implementation of the WS-Eventing specification (WP3)

3.2.4.2 Capability *IRetrieveContext::subscribe()*

int subscribe(string ContextInfoDescription, string ContextSubscriptionReference) - the arguments of this method might change, depending on the .NET implementation of the WS Eventing specification (WP3)

3.2.4.3 Capability *IRetrieveContext::unsubscribe()*

bool unsubscribe(int SubscriptionID) - the arguments of this method might change, depending on the .NET implementation of the WS Eventing specification (WP3)

3.2.4.4 Capability *IRetrieveContext::query()*

DataSet query(string ContextQueryExpression) – this method provides the capability to query the context history database. The query method has the SQL query string as an argument (ContextQueryExpression), and the method returns the result of the query as a DataSet.

3.2.5 Dynamic Modeler

The Dynamic Modeler component offers the external interface *IModelMultimed* for dynamic modeling of user preferences in such domains as multimedia retrieval, information retrieval and other similar domains. Dynamic Modeler can be used either under full application control or in automatic mode of action.

In case of full application control application should provide log file in a form context - item name – positive/ negative label and to call corresponding functions to get CBR-based recommendations, SVM-based recommendations or both. Applications should also call function which initiates SVM training (CBR does not require training).

If applications needs automatic Dynamic Modelling, it should only provide log files in a form timestamp – user action (such as switching on movie, stopping movie, pausing etc) and to set parameters which would tell how to label examples (that is, to specify what is positive and what is negative example of implicit user feedback). It is also desirable to provide explicit user feedback whenever possible (via FeedbackAnalyzer) because explicit user feedback is more reliable than implicit feedback. Dynamic Modelling (with the help of FeedbackAnalyzer) will use context data of ContextInterpreter component of CMS and application log files for deriving positive and negative examples of implicit feedback and re-training classifiers. When

applications require recommendations, this recommendations will be provided as a weighted sum of recommendations from StaticModeler, CBR and SVM, where weights depend on past prediction accuracy of these components.

All user preferences are treated as context-dependent, where context can be current or future events (personal events, events of other family members, more generic events as public holidays or Olympic Games), time, presence of other people and whatever other factors applications consider important. Context-based recommendations are performed by both SVM and CBR.

Additionally, Dynamic Modelling of user preferences provides the capability to get context-based recommendations (the items which were previously retrieved in similar contexts) and so-called frequency-based recommendations: the items which were retrieved most frequently in the household, no matter in which contexts they were retrieved. The Dynamic Modeler component also provides the capability to retrieve N last items in the history, which is needed when an application should not suggest same items soon after they were used previous time. All the capabilities offered through this interface are described in the following section, including detailed information for the implemented ones.

3.2.5.1 Capability IModelMultimed::

GetContextBasedDynamicSuggestionsFromPartOfLogFile ()

```
string GetContextBasedDynamicSuggestionsFromPartOfLogFile(string LogFileName, string
MetadataFileName, string CurrentContext, string SpecialWeights, double
RankThresholdForReturnCases, int NumOfCasesToProcess)
```

This function assumes that log file contains items (for example, "apple pie" or "comedy" or "The Mask"), and returns ranked and sorted (by rank) list of items from the log file. For example: "comedy", rank 0.5, confidence 0.2 2)"opera", rank 0.5, confidence 0.1 3)"thriller", rank 0.1, confidence 0.7, where item rank is based on similarity of the sets of context descriptors. The similarity is calculated between the set of context descriptors contained in CurrentContext input parameter, and a set of context descriptors of each item retrieval case in application log file. If item was retrieved several times, its overall rank is calculated as an average of contexts similarities of all these cases.

If MetadataFileName is not empty, this function will use log file for learning which metadata items are desirable in each context, e.g., if log file contains names of movies, Dynamic Modeler will build models for genres, actors etc.

This function assumes that set of context descriptors means AND concatenation: Monday AND Evening AND User1 AND User2. Each context descriptor is a string; strings should match exactly (e.g., similarity between "userID: 1" and "user ID: 1" is 0). Thus, application developers should take care that context descriptors in CurrentContext are from the same ontology as context descriptors in the log file

When item rank is calculated, all context descriptors are treated as equally important by default. The input parameter SpecialWeights allows to assign different importance to different context descriptors. For example, if preferences of some particular user are most important, the following SpecialWeight can be assigned:

```
<key id="userID: 1">
  <valueset>
    <value>10</value>
  </valueset>
</key>
```

In this case all items, retrieved by the user1, will be on the top of recommendation list. Other context parameters will have smaller input to the rank of items.
All context descriptors, not listed in SpecialWeights input parameter, are assigned the default weight = 1.

Confidence of recommendation is calculated as Number Of Occurances Of Item in the Processes History, divided by Total Number Of Items In The Processed History
Confidence parameter can be used for automatic update of user profile (if the confidence is higher than a predefined threshold, update the profile; otherwise wait). Confidence is also used as an additional input to the sorting procedure: if two items have the same context-based rank, the item with a higher confidence will be put on top of the item with a lower confidence.

Ranking and sorting of items does not make any assumptions regarding whether recent items can be of higher or lower interest to the users. However, the items are returned together with the context of last retrieval (only the last context!), so application developer can decide, whether an item should be recommended 2 times in a row or not.

Function output: XML string

```
<key id="comedy">
  <valueset>
    <value>0.865</value>
    <confidence>.167</confidence>
    <parameter id="context">
      <value>userID: 1␣Sunday␣Evening</value>
    </parameter>
  </valueset>
</key>
<key id="thriller">
  <valueset>
    <value>0.667</value>
    <confidence>.167</confidence>
    <parameter id="context">
      <value>userID: 1␣userID: 2␣Monday␣Afternoon</value>
    </parameter>
  </valueset>
</key>
```

Context descriptors are temporally concatenated with ␣ symbol.

Function input parameters:

1. string LogFileName: full path of file where application log is stored. The format should be the following:

```
s1:
contextDescriptor_1 (e.g., "Monday")
contextDescriptor_2 (e.g., "evening")
...
contextDescriptor_n (e.g., "userID: 1")
s2:
name_of_user_choice (e.g., "Bolero") and positive/negative label
```

IMPORTANT: if two items are listed under "s2:", they are treated as separate cases. Label 0 will mean that example is definitely negative, label 101 will mean positive example, and value in between will allow to treat example as positive or negative depending on a threshold (default threshold value is 80). Negative examples are used only for SVM training, however, the functions described in this section use only positive examples because in our tests prediction accuracy of CBR was same and even better if it used only positive examples. Thus, functions described in this section consider an example as positive if no label is provided.

2. string CurrentContext: a string which contains a set of context descriptors. As a temporal solution context descriptors in the set are concatenated with α symbol, and it is assumed to be AND concatenation (that is, user1 α user2 CurrentContext parameter denotes presence of both users).

3. string SpecialWeights: as stated above, an XML string which lists most important context descriptors and their weights

4. double RankThresholdForReturnCases: the item is recommended only if its rank exceeds RankThresholdForReturnCases

4. int NumOfCasesToProcess: number which states how many last cases in the log files are included into processing. All previous cases are ignored.

3.2.5.2 Capability IModelMultimed::

GetContextBasedDynamicSuggestionsFromLogFile ()

string GetContextBasedDynamicSuggestionsFromLogFile(string LogFileName, string MetadataFileName, string CurrentContext, string SpecialWeights, double RankThresholdForReturnCases): the overloaded version of the function GetContextBasedDynamicSuggestionsFromPartOfLogFile, but this function always processes the whole log file

3.2.5.3 Capability IModelMultimed::

GetNContextBasedDynamicSuggestionsFromPartOfLogFile ()

string GetNContextBasedDynamicSuggestionsFromPartOfLogFile(string LogFileName, string MetadataFileName, string CurrentContext, string SpecialWeights, int NumOfRecommendationsToReturn, int NumOfCasesToProcess): the overloaded version of the function GetContextBasedDynamicSuggestionsFromPartOfLogFile, but this function always returns NumOfRecommendationsToReturn number of top-ranked items (e.g., top ten recommendations), independently on whether ranks of recommended items are in fact low, or whether the ranks of not recommended items are equal to ranks of recommended items. The function returns less than NumOfRecommendationsToReturn number of items only if ranks of not included items do not exceed zero.

3.2.5.4 Capability IModelMultimed::

GetNContextBasedDynamicSuggestionsFromLogFile ()

string GetNContextBasedDynamicSuggestionsFromLogFile(string LogFileName, string MetadataFileName, string CurrentContext, string SpecialWeights, int NumOfRecommendationsToReturn): the overloaded version of the function GetNContextBasedDynamicSuggestionsFromPartOfLogFile, but this function always processes the whole file.

3.2.5.5 Capability IModelMultimed::**GetFrequencyBasedDynamicSuggestionsFromPartOfLogFile ()**

string GetFrequencyBasedDynamicSuggestionsFromPartOfLogFile(string LogFileName, string MetadataFileName, int NumOfFavorites, int NumOfCasesToProcess)

This function assumes the same format of the log file as GetContextBasedDynamicSuggestions functions, but it returns items frequently encountered in the log file, independently on contexts when these items were retrieved.

The items are sorted based on their ranks, but here item rank is calculated as Number Of Occurances Of Item in the processed history, e.g., "comedy", rank 5 ; "opera", rank 2 /This function does not make any assumptions regarding whether recent items can be of higher or lower interest to the users, but the items are returned together with the context of last retrieval (only the last!).

Function output: XML string

```
<key id="comedy">
  <valueset>
    <value>5</value>
    <parameter id="context">
      <value>userID: 1␣Sunday␣Evening</value>
    </parameter>
  </valueset>
</key>
<key id="thriller">
  <valueset>
    <value>2</value>
    <parameter id="context">
      <value>userID: 1␣userID: 2␣Monday␣Afternoon</value>
    </parameter>
  </valueset>
</key>
```

Context descriptors are temporally concatenated with ␣ symbol.

Function input parameters:

1. string LogFileName: full path of file where application log is stored. The format should be the following:

```
s1:
contextDescriptor_1 (e.g., "Monday")
contextDescriptor_2 (e.g., "evening")
...
contextDescriptor_n (e.g., "userID: 1")
s2:
name_of_user_choice (e.g., "Bolero")
```

IMPORTANT: if two items are listed under "s2:", they are treated as separate cases.

2. int NumOfFavorites: an item is included in the list only if its Number Of Occurances in the processed history is not less than NumOfFavorites parameter

3. int NumOfCasesToProcess: number which states how many last cases in the log files are included into processing. All previous cases are ignored.

3.2.5.6 Capability IModelMultimed::

GetFrequencyBasedDynamicSuggestionsFromLogFile ()

string GetFrequencyBasedDynamicSuggestionsFromLogFile(string LogFileName, string MetadataFileName, int NumOfFavorites): the overloaded version of the function GetFrequencyBasedDynamicSuggestionsFromPartOfLogFile, but this function always processes the whole log file

3.2.5.7 Capability IModelMultimed::

GetNFrequencyBasedDynamicSuggestionsFromPartOfLogFile ()

string GetNFrequencyBasedDynamicSuggestionsFromPartOfLogFile(string LogFileName, string MetadataFileName, int NumOfRecommendationsToReturn, int NumOfCasesToProcess): the overloaded version of the function GetFrequencyBasedDynamicSuggestionsFromPartOfLogFile, but this function always returns NumOfRecommendationsToReturn number of most frequently encountered items (e.g., top ten recommendations), independently on whether frequencies of recommended items are in fact low, or whether the frequencies of not recommended items are equal to frequencies of recommended items. The function returns less than NumOfRecommendationsToReturn number of items only if frequencies of not included items are equal to zero.

3.2.5.8 Capability IModelMultimed::

GetNFrequencyBasedDynamicSuggestionsFromLogFile ()

string GetNFrequencyBasedDynamicSuggestionsFromLogFile(string LogFileName, string MetadataFileName, int NumOfRecommendationsToReturn): the overloaded version of the function GetNFrequencyBasedDynamicSuggestionsFromPartOfLogFile, but this function always processes the whole file.

3.2.5.9 Capability IModelMultimed:: GetNLastItemsFromLogFile ()

string GetNLastItemsFromLogFile(string LogFileName, string MetadataFileName, int NumOfLastItems) returns NumOfLastItems from the log file, starting from the last case.

Although the term "rank" is still used in the output string, this function does not rank items. Instead of ranks, the number of the case in the history is returned, for example: 1) "comedy", rank = 155; "thriller", rank = 154

The items are returned together with the context of their retrieval.

Function output: XML string

```
<key id="comedy">
  <valueset>
    <value>155</value>
    <parameter id="context">
      <value>userID: 1#Sunday#Evening</value>
    </parameter>
  </valueset>
</key>
```

```

<key id="thriller">
  <valueset>
    <value>154</value>
    <parameter id="context">
      <value>userID: 1⌘userID: 2⌘Monday⌘Afternoon</value>
    </parameter>
  </valueset>
</key>

```

Context descriptors are temporally concatenated with ⌘ symbol.

Function input parameters:

1. string LogFileName: full path of file where application log is stored. The format should be the following:

```

s1:
contextDescriptor_1 (e.g., "Monday")
contextDescriptor_2 (e.g., "evening")
...
contextDescriptor_n (e.g., "userID: 1")
s2:
name_of_user_choice (e.g., "Bolero")

```

IMPORTANT: if two items are listed under "s2:", they are treated as separate cases.

2. int NumOfLastItems: this number of items is returned

3.2.6 Multi-Profile Aggregator

The Multi-Profile aggregator component offers the external interface `IMultiProfileAggregation` for aggregation of preferences of multiple users in such domains as multimedia retrieval, information retrieval and other similar domains. The aggregation of preferences is based, first, on the observations which items multiple users retrieve when they are all gathered together: this is a dynamic modeling part of multi-profile aggregation. The dynamic modeling part of Multi-Profile aggregator provides the capability to get context-based recommendations (the items which were previously retrieved by multiple users in similar contexts) and so-called frequency-based recommendations: the items which were retrieved most frequently by this set of multiple users, no matter in which contexts they were retrieved. The dynamic modeling part of Multi-Profile aggregator component also provides the capability to retrieve N last items in the history, retrieved by this set of multiple users, which is needed when an application should not suggest same items soon after they were used previous time. Second, the aggregation of static user profiles (the profiles acquired explicitly) will be implemented later and will provide the capability to retrieve the union of preferences of multiple users (e.g., for an application which will download all items of possible user interest to a device to be taken to a trip) and intersection of preferences.

All the capabilities offered through this interface are described in the following section, including detailed information for the implemented ones.

3.2.6.1 Capability IMultiProfileAggregation::**GetMultiProfileContextBasedDynamicSuggestionsFromPartOfLogFile ()**

string GetMultiProfileContextBasedDynamicSuggestionsFromPartOfLogFile(string LogFileName, string MetadataFileName, string userIDs, string CurrentContext, string SpecialWeights, double RankThresholdForReturnCases, int NumOfCasesToProcess)

This function assumes that log file contains items (for example, "apple pie" or "comedy"), and returns ranked and sorted (by rank) list of items from the log file. For example: "comedy", rank 0.5, confidence 0.2 2)"opera", rank 0.5, confidence 0.1 3)"thriller", rank 0.1, confidence 0.7, where item rank is based on similarity of the sets of context descriptors. The similarity is calculated between the set of context descriptors contained in CurrentContext input parameter, and a set of context descriptors of each item retrieval case in application log file. If item was retrieved several times, its overall rank is calculated as an average of contexts similarities of all these cases. Items which were retrieved when not all multiple users, listed in the string userIDs input parameter, were present, are also retrieved by this function; however, their ranks are lower than ranks of items which were retrieved when all users were present. CurrentContext parameter should not include userIDs; it is used when e.g. application wants to rank higher items which were retrieved on Friday evening.

If MetadataFileName is not empty, this function will use log file for learning which metadata items are desirable in each context, e.g., if log file contains names of movies, Dynamic Modeler will build models for genres, actors etc.

This function assumes that set of context descriptors means AND concatenation: Friday AND Evening. Each context descriptor is a string; strings should match exactly (e.g., similarity between "userID: 1" and "user ID: 1" is 0). Thus, application developers should take care that context descriptors in CurrentContext are from the same ontology as context descriptors in the log file, and that userIDs match those stored in a log file.

When item rank is calculated, all context descriptors are treated as equally important by default. The input parameter SpecialWeights allows to assign different importance to different context descriptors. For example, if preferences for Friday are most important, the following SpecialWeight can be assigned:

```
<key id="Friday">
  <valueset>
    <value>10</value>
  </valueset>
</key>
```

In this case all items, retrieved on Friday, will be on the top of recommendation list. Other context parameters will have smaller input to the rank of items.

All context descriptors, not listed in SpecialWeights input parameter, are assigned the default weight = 1.

Confidence of recommendation is calculated as Number Of Occurances Of Item in the Processes History, divided by Total Number Of Items In The Processed History. Processed History includes items viewed when not all multiple users were gathered together. Thus, if some item was viewed two times, first when not all multiple users were not present; and second, when they were present; the item rank will be lower, but the confidence in its rank will be higher than for the item which was retrieved only once when all multiple users were present.

Confidence parameter can be used for automatic update of user profile (if the confidence is higher than a predefined threshold, update the profile; otherwise wait). Confidence is also

used as an additional input to the sorting procedure: if two items have the same context-based rank, the item with a higher confidence will be put on top of the item with a lower confidence.

Ranking and sorting of items does not make any assumptions regarding whether recent items can be of higher or lower interest to the users. However, the items are returned together with the context of last retrieval (only the last context!), so application developer can decide, whether an item should be recommended 2 times in a row or not.

Function output: XML string

```
<key id="comedy">
  <valueset>
    <value>0.865</value>
    <confidence>.167</confidence>
    <parameter id="context">
      <value>userID: 1⌘ userID: 2⌘Sunday⌘Evening</value>
    </parameter>
  </valueset>
</key>
<key id="thriller">
  <valueset>
    <value>0.367</value>
    <confidence>.167</confidence>
    <parameter id="context">
      <value>userID: 1⌘userID: 2⌘Monday⌘Afternoon</value>
    </parameter>
  </valueset>
</key>
```

Context descriptors are temporally concatenated with ⌘ symbol.

Function input parameters:

1. string LogFileName: full path of file where application log is stored. The format should be the following:

```
s1:
contextDescriptor_1 (e.g., "Monday")
contextDescriptor_2 (e.g., "evening")
...
contextDescriptor_n (e.g., "userID: 1")
s2:
name_of_user_choice (e.g., "Bolero")
```

IMPORTANT: if two items are listed under "s2:", they are treated as separate cases.

2. string userIDs: string that lists IDs for multi-profile aggregation in XML format:

```
<key id="user1"></key><key id="user2"></key>
```

3. string CurrentContext: a string which contains a set of context descriptors. As a temporal solution context descriptors in the set are concatenated with ⌘ symbol, and it is assumed to be AND concatenation (that is, Friday⌘Monday CurrentContext parameter is nonsense).

4. string SpecialWeights: as stated above, an XML string which lists most important context descriptors and their weights

5. double RankThresholdForReturnCases: the item is recommended only if its rank exceeds RankThresholdForReturnCases

6. int NumOfCasesToProcess: number which states how many last cases in the log files are included into processing. All previous cases are ignored.

3.2.6.2 Capability IMultiProfileAggregation::

GetMultiProfileContextBasedDynamicSuggestionsFromLogFile ()

string GetMultiProfileContextBasedDynamicSuggestionsFromLogFile(string LogFileName, string MetadataFileName, string userIDs, string CurrentContext, string SpecialWeights, double RankThresholdForReturnCases): the overloaded version of the function GetMultiProfileContextBasedDynamicSuggestionsFromPartOfLogFile, but this function always processes the whole log file

3.2.6.3 Capability IMultiProfileAggregation::

GetMultiProfileNContextBasedDynamicSuggestionsFromPartOfLogFile ()

string GetMultiProfileNContextBasedDynamicSuggestionsFromPartOfLogFile(string LogFileName, string MetadataFileName, string userIDs, string CurrentContext, string SpecialWeights, int NumOfRecommendationsToReturn, int NumOfCasesToProcess): the overloaded version of the function GetMultiProfileContextBasedDynamicSuggestionsFromPartOfLogFile, but this function always returns NumOfRecommendationsToReturn number of top-ranked items (e.g., top ten recommendations), independently on whether ranks of recommended items are in fact low, or whether the ranks of not recommended items are equal to ranks of recommended items. The function returns less than NumOfRecommendationsToReturn number of items only if ranks of not included items do not exceed zero.

3.2.6.4 Capability IMultiProfileAggregation::

GetMultiProfileNContextBasedDynamicSuggestionsFromLogFile ()

string GetMultiProfileNContextBasedDynamicSuggestionsFromLogFile(string LogFileName, string MetadataFileName, string userIDs, string CurrentContext, string SpecialWeights, int NumOfRecommendationsToReturn): the overloaded version of the function GetMultiProfileNContextBasedDynamicSuggestionsFromPartOfLogFile, but this function always processes the whole file.

3.2.6.5 Capability IMultiProfileAggregation::

GetMultiProfileFrequencyBasedDynamicSuggestionsFromPartOfLogFile ()

string GetMultiProfileFrequencyBasedDynamicSuggestionsFromPartOfLogFile(string LogFileName, string MetadataFileName, string userIDs, int NumOfFavorites, int NumOfCasesToProcess)

This function assumes the same format of the log file as GetMultiProfileContextBasedDynamicSuggestions functions, but it returns items frequently encountered in the log file when all multiple users were present (and only when they all were present!), independently on contexts when these items were retrieved.

The items are sorted based on their ranks, but here item rank is calculated as Number Of Occurrences Of Item among cases when all multiple users were present, e.g., "comedy", rank 5 ; "opera", rank 2

This function does not make any assumptions regarding whether recent items can be of higher or lower interest to the users, but the items are returned together with the context of last retrieval (only the last!).

Function output: XML string

```
<key id="comedy">
  <valueset>
    <value>5</value>
    <parameter id="context">
      <value>userID: 1␣userID: 2␣Sunday␣Evening</value>
    </parameter>
  </valueset>
</key>
<key id="thriller">
  <valueset>
    <value>2</value>
    <parameter id="context">
      <value>userID: 1␣userID: 2␣Monday␣Afternoon</value>
    </parameter>
  </valueset>
</key>
```

Context descriptors are temporally concatenated with ␣ symbol.

Function input parameters:

1. string LogFileName: full path of file where application log is stored. The format should be the following:

```
s1:
contextDescriptor_1 (e.g., "Monday")
contextDescriptor_2 (e.g., "evening")
...
contextDescriptor_n (e.g., "userID: 1")
s2:
name_of_user_choice (e.g., "Bolero")
```

IMPORTANT: if two items are listed under "s2:", they are treated as separate cases.

2. string userIDs: string that lists IDs for multi-profile aggregation in XML format:

```
<key id="user1"></key><key id="user2"></key>
```

3. int NumOfFavorites: an item is included in the list only if its Number Of Occurances in the processed history is not less than NumOfFavorites parameter

4. int NumOfCasesToProcess: number which states how many last cases in the log files are included into processing. All previous cases are ignored.

3.2.6.6 Capability IMultiProfileAggregation::

GetMultiProfileFrequencyBasedDynamicSuggestionsFromLogFile ()

string GetMultiProfileFrequencyBasedDynamicSuggestionsFromLogFile(string LogFileName, string MetadataFileName, string userIDs, int NumOfFavorites): the overloaded version of the

function `GetMultiProfileFrequencyBasedDynamicSuggestionsFromPartOfLogFile`, but this function always processes the whole log file

3.2.6.7 Capability `IMultiProfileAggregation::`

`GetMultiProfileNFrequencyBasedDynamicSuggestionsFromPartOfLogFile ()`

`string GetMultiProfileNFrequencyBasedDynamicSuggestionsFromPartOfLogFile(string LogFileName, string MetadataFileName, string userIDs, int NumOfRecommendationsToReturn, int NumOfCasesToProcess)`: the overloaded version of the function `GetMultiProfileFrequencyBasedDynamicSuggestionsFromPartOfLogFile`, but this function always returns `NumOfRecommendationsToReturn` number of most frequently encountered items (e.g., top ten recommendations) among items retrieved when all multiple users were present, independently on whether frequencies of recommended items are in fact low, or whether the frequencies of not recommended items are equal to frequencies of recommended items. The function returns less than `NumOfRecommendationsToReturn` number of items only if frequencies of not included items are equal to zero.

3.2.6.8 Capability `IMultiProfileAggregation::`

`GetMultiProfileNFrequencyBasedDynamicSuggestionsFromLogFile ()`

`string GetMultiProfileNFrequencyBasedDynamicSuggestionsFromLogFile(string LogFileName, string MetadataFileName, string userIDs, int NumOfRecommendationsToReturn)`: the overloaded version of the function `GetMultiProfileNFrequencyBasedDynamicSuggestionsFromPartOfLogFile`, but this function always processes the whole file.

3.2.6.9 Capability `IMultiProfileAggregation::` **`GetMultiProfileNLastItemsFromLogFile ()`**

`string GetMultiProfileNLastItemsFromLogFile(string LogFileName, string MetadataFileName, string userIDs, int NumOfLastItems)` returns `NumOfLastItems` from the log file, retrieved when all multiple users were present, and starting from the last case.

Although the term "rank" is still used in the output string, this function does not rank items. Instead of ranks, the number of the case in the history is returned, for example: 1) "comedy", rank = 155; "thriller", rank = 150. Ranks in this function are not necessarily consecutive because cases when all multiple users were present are not consecutive in the log file.

The items are returned together with the context of their retrieval.

Function output: XML string

```
<key id="comedy">
  <valueset>
    <value>155</value>
    <parameter id="context">
      <value>userID: 1␣ userID: 2␣Sunday␣Evening</value>
    </parameter>
  </valueset>
</key>
<key id="thriller">
```

```

<valueset>
  <value>150</value>
  <parameter id="context">
    <value>userID: 1⌘userID: 2⌘Monday⌘Afternoon</value>
  </parameter>
</valueset>
</key>

```

Context descriptors are temporally concatenated with ⌘ symbol.

Function input parameters:

1. string LogFileName: full path of file where application log is stored. The format should be the following:

```

s1:
contextDescriptor_1 (e.g., "Monday")
contextDescriptor_2 (e.g., "evening")
...
contextDescriptor_n (e.g., "userID: 1")
s2:
name_of_user_choice (e.g., "Bolero")

```

IMPORTANT: if two items are listed under "s2:", they are treated as separate cases.

2. string userIDs: string that lists IDs for multi-profile aggregation in XML format:
 <key id="user1"></key><key id="user2"></key>

3. int NumOfLastItems: this number of items is returned

3.2.6.10 Capability IMultiProfileAggregation:: getMultiProfileUnionOfStaticPrefs ()

string getMultiProfileUnionOfStaticPrefs(string settingID, string userIDs, string CurrentContext, string SpecialWeights, double ContextSimilarityThreshold) returns maximum preference value of this SettingID among the users' Static Profiles

Input:

settingID – same as in UMPS Reasoning Module functions

userIDs: string that lists IDs for multi-profile aggregation

format: <key id="user1"></key><key id="user2"></key><key id="user3"></key>

CurrentContext: Context descriptors in CurrentContext should be from the same ontology as in the log file, because Rank is calculated by matching context descriptor strings

If CurrentContext string is empty string, this function returns max preference value among generic preference values of users

SpecialWeights parameter is needed for assigning higher importance for some context parameters, for example, if "Sunday" and "Saturday" contexts are very important for an application, application can have following SpecialWeights:

```
/*
  <key id>="Sunday">
    <valueset>
      <value>100</value>
    </valueset>
  </key>
  <key id>="Saturday">
    <valueset>
      <value>100</value>
    </valueset>
  </key>
*/
//all context descriptors, not listed in SpecialWeights, are assigned the weight = 1
```

ContextSimilarityThreshold: if the degree of similarity between query context and contexts, stored in the user profile, does not exceed ContextSimilarityThreshold, a generic preference value is taken; otherwise context-dependent preference value is taken

3.2.6.11 Capability **IMultiProfileAggregation::getMultiProfileMinOfStaticPrefs ()**

string getMultiProfileMinOfStaticPrefs(string settingID, string userIDs, string CurrentContext, string SpecialWeights, double ContextSimilarityThreshold) returns minimum preference value of this SettingID among the users' Static Profiles; this is so-called "Least Misery Strategy": the group is as happy as its least happy member

Inputs and outputs same as in getMultiProfileUnionOfStaticPrefs function. If CurrentContext string is empty string, this function returns min preference value among generic preference values of users

3.2.6.12 Capability **IMultiProfileAggregation::getMultiProfileWeightedAverageOfStaticPrefs ()**

string getMultiProfileWeightedAverageOfStaticPrefs(string settingID, string userIDsAndWeights, string CurrentContext, string SpecialWeights, double ContextSimilarityThreshold) returns weighted average value of this SettingID among the users' Static Profiles; this is "Average Strategy"

Input:

settingID – same as in UMPS Reasoning Module functions

userIDsAndWeights: string that lists IDs for multi-profile aggregation and the corresponding weights in the same format as SpecialWeights parameter, but in the string userIDs ALL USERS should be listed

Normal practice in calculating Weighted Average is that sum of weights should be equal to 1

Format:

```
<key id>="user1">
  <valueset>
    <value>0.7</value>
```

```

    </valueset>
  </key>
  <key id="user2">
    <valueset>
      <value>0.3</value>
    </valueset>
  </key>

```

CurrentContext: Context descriptors in CurrentContext should be from the same ontology as in the log file, because Rank is calculated by matching context descriptor strings.

If CurrentContext string is empty string, this function returns weighted average of preference values among generic preference values of users

SpecialWeights parameter is needed for assigning higher importance for some context parameters, for example, if "Sunday" and "Saturday" contexts are very important for an application, application can have following SpecialWeights:

```

  <key id="Sunday">
    <valueset>
      <value>100</value>
    </valueset>
  </key>
  <key id="Saturday">
    <valueset>
      <value>100</value>
    </valueset>
  </key>

```

All context descriptors, not listed in SpecialWeights, are assigned the weight = 1

ContextSimilarityThreshold: if the degree of similarity between query context and contexts, stored in the user profile, does not exceed ContextSimilarityThreshold, a generic preference value is taken; otherwise context-dependent preference value is taken

3.2.6.13 Capability **IMultiProfileAggregation::getMultiProfileWeightedAverageWithoutMiseryOfStaticPrefs ()**

string getMultiProfileWeightedAverageWithoutMiseryOfStaticPrefs(string settingID, string userIDsAndWeights, string CurrentContext, string SpecialWeights, double ContextSimilarityThreshold, double PrefsMiseryThreshold) returns weighted average value of this SettingID among the users' Static Profiles, if preferences of ALL users exceed PrefsMiseryThreshold; otherwise it returns minimum of preference values among the users
This is so-called "Average without Misery Strategy"

Except for the addition of PrefsMiseryThreshold input parameter, this function has the same inputs as getMultiProfileWeightedAverageOfStaticPrefs function. If CurrentContext string is empty string, this function returns "weighted average without misery" of preference values among generic preference values of users

3.2.7 Functionality used by both DynamicModeler and Multi-Profile Aggregator

Since presence of people can be seen also as context, preferences of a single user and preferences of a multi-user environment can be learned in a same way. We provide, first,

functions to train SVM (Support Vector Machines) and to get SVM estimates of users' preferences; and second, functions to activate fully automatic dynamic modeling with CBR, SVM and explicitly acquired user preferences and to get combined estimates of users' preferences.

3.2.7.1 Capability IModelMultimed:: trainSVM(...)

```
void trainSVM(string MetadataTreeFile, string ContextTreeFile, string ContextsToUse, string
HistoryFileName, string Domain, string ModelsDir, float PositiveThreshold, float PenaltyPos,
float PenaltyNeg, float PenaltyExplicitPos, float PenaltyExplicitNeg, bool UseExtraZeros)
```

This function allows applications to control SVM training.

We use SVM implementation in TORCH library of machine-learning methods. SVM is trained to provide recommendations on each metadata item (each genre, each actor etc) separately, that is, SVM builds distinct models for all items: own model for predicting whether the user is interested in "comedy" genre or not in different contexts; own model for predictions regarding "thriller"; own model for predictions regarding "actor1" and so on. Models are produced only if training data contains more than ten positive and ten negative examples for this metadata item (in our tests we used for predictions also SVM models trained with only 2 positive examples, but this approach leads to significant increase in number of models without significant improvements in prediction accuracy).

Function parameters:

UseExtraZeros: SVM builds decision surface between positive and negative examples, in a "one against others" way. Generally training can be performed separately for each metadata item (e.g., each genre, each actor etc) if sufficient number of positive and negative examples for each item is provided. For example, Feedback Analyzer can observe that the users always watch comedy from the beginning until the end on Fridays, but they never choose comedy from recommendation list on Saturdays and choose drama instead. This observation provides both positive and negative examples for comedies viewing. In practice, however, such case is not common. In the example above, SVM will know that movies of drama genre are viewed on Saturdays, but it will not know when drama is not desirable genre. Thus, common way to create negative examples is to make different genres to compete with each other, that is, if comedy was viewed, than drama was not viewed. This way we get positive example for one metadata item and negative examples for all other items under common parent in ontology tree if UseExtraZeros parameter is true. If UseExtraZeros parameter is false, SVM will learn models for each metadata item only when sufficient number of examples for this particular item is gathered.

MetadataTreeFile: content ontology file. When extra negative examples are created from metadata items, not selected for viewing, they are created only for items under common parent in ontology tree. For example, genres compete with each other in a "one against all" way, countries of movie origin can compete with each other in a same way, but competition of genres with countries of movie origin does not make sense. When new examples arrive, it is sufficient to re-train only models under the same parent in ontology tree. Because of this, SVM training requires carefully designed metadata tree: if there are too many leaf nodes under same parent, re-training of models will take longer time. If there are too few leaf nodes, SVM might not get sufficient number of negative examples.

domain: name of a parent node which child nodes will compete with each other in training. If domain is "genre" –all genres under "genre" node will compete with each other, e.g., "historical movies" will compete with comedies, tragedies etc. If domain is "historical movies", then e.g. "movies about ancient history" will compete with "movies about 2d world war" and so on.

ContextsToUse: list of semantic context values which application considers important for learning, separated with whitespaces, e.g., "workday weekend user1 user2 user3". In SVM

training, each line of train file consists of context predicate and positive/ negative label (0 or 1). Context predicate is also a list of zeros and ones, where ones appear when context value equals to the value specified in “ContextsToUse” string, and zeros appear if value is different. ContextsToUse string “workday weekend user1 user2 user3” will result e.g. in a context predicate 1 0 1 0 0 for context set “Monday, user1” because Monday is a workday and because user1 is present while other users are absent.

ContextTreeFile: context ontology file. Context ontology tree is needed because application log files might contain too detailed information. For example, our log files contain “Monday”, “Tuesday” etc data, while in some cases it is only important to distinguish between workdays and weekends. Context ontology tree helps to change level of details. In the example above, we got first “one” value (in place of “workday”) because context ontology knew that Monday is a workday.

HistoryFileName: file which contains contexts and user choices, in a same format as history files used by CBR and described above. However, CBR does not necessarily require negative examples (in our tests prediction accuracy of CBR was same and even better if it used only positive examples), while for SVM it is essential to get both positive examples. Thus, for SVM history file must contain also a label whether an example is positive or negative, while for CBR such label is optional (an example is considered positive if no label is provided).

PositiveThreshold: if label value is greater than PositiveThreshold value, an example is positive, otherwise negative. Value 0 denotes explicit negative examples, value 101 denotes explicit positive examples.

ModelsDir: directory where SVM train files and models should be stored. When training is finished and SVM makes recommendations, it looks through all created models in order to find the most appropriate model for the current context and metadata item; thus, it might make sense to store different sets of models in different directories to make this search for the appropriate model faster if overall number of models is large.

PenaltyPos, PenaltyNeg, PenaltyExplicitPos, PenaltyExplicitNeg: during SVM training different penalties for misclassification of different classes of examples can be assigned. PenaltyPos / PenaltyNeg are penalties for misclassification of implicit feedback, and PenaltyExplicitPos / PenaltyExplicitNeg are penalties for misclassification of explicit feedback (and should be higher). Choice of penalties should depend on number of examples, overall number of metadata items and on how well the model can be learned in ideal case (e.g., SVM can not learn user model if the user makes choices randomly). Default penalties are 100 for all examples, and in our tests with real life TV data we observed that setting too high penalties (let's say, higher than 700) for some families can make training very slow or result in the situation when training does not converge.

3.2.7.2 Capability IModelMultimed:: getSVMrank(...)

string getSVMrank (string ModelsDir, string MetadataItem, string MetadataTreeFile, string CurrentContext)

CurrentContext: list of semantic context values, separated with whitespaces, e.g., “Monday evening user1 user2”

MetadataItem: SettingID (ontology term which rank is needed), e.g., “comedy” or ontology term for which list of ranks is needed, e.g., “genre” – in this case list of ranks of all child nodes will be returned.

MetadataTreeFile: same file as used for SVM training. This parameter is not used if application is interested in rank of metadata item for which SVM model exists, e.g., if application is interested in SVM rank for “comedy”. If application is interested in list of ranks for all genres, then metadata tree will be used for finding child nodes of “genre” node (only two-level tree is supported)

ModelsDir: directory where SVM stored models during training.

Return string:

```
<key id="comedy">
  <valueset>
    <value>0.865</value>
  </valueset>
</key>
```

3.2.7.3 Capability IModelMultimed:: startLearning(...)

```
bool startLearning ( string configurationFile, string ContextInterpreterLogFile, string
applicationLogFile, string MetadataTreeFile, string ContextTreeFile, string ContextsToUse,
string MetadataInfoFile)
```

This function will initiate automatic dynamic modelling process which consists of the following steps: FeedbackAnalyzer will check application log files and context data logs of ContextInterpreter and create a history file in a format "context-user choice-label", required by dynamic modelling components. FeedbackAnalyzer will also check Feedback log files for explicit user feedback and will add to history examples of explicit feedback. Then history will be used by SVM and CBR for estimation of user preference value (we also call it rank), and the overall user preference value will be calculated as weighted sum of Static Model, CBR and SVM values.

In the beginning of learning process it is only possible to return user preference value retrieved from static model, but when sufficient number of examples is gathered, also SVM and CBR estimates of user preference value can be used. Since prediction accuracy of each model is user-dependent (in our tests for some users SVM prediction accuracy was significantly better, while for other users it was vice versa; and for some users none of classifiers was accurate enough), combined user preference value (rank) is calculated as weighted sum of ranks of three components of user model, where weights depend on past prediction accuracy. Thus, each time when application queries for user preferences, FeedbackAnalyzer will store the query and the preference estimates by three UMPS components: Static Model, CBR dynamic model and SVM dynamic model. Each time when user feedback (implicit or explicit) is estimated, Feedback Analyzer will evaluate prediction accuracy of these three models and re-calculate weights.

Since training SVM requires time if number of metadata items is large and misclassification penalties are high, re-training takes place only at night (not after a new case is added to the history)

Function input:

configurationFile – path to the file which contains parameters required by FeedbackAnalyzer functions getPositiveExamples and getNegativeExamples (see above) and also penalties for tuning SVM training. If configurationFile is not provided, or if not all parameters are specified in it, default parameters will be used.

Format of configuration file is following:

```
<type>PenaltyPos</type><value>300</value><type>PenaltyExplicitPos</type><value>500</v
alue><type>UseExtraZeros</type><value>true</value><type>PositivePercent</type><value>
60</value>
```

Example of configuration file with default parameters is provided together with software.

This function should be called only after some application log files

ContextInterpreterLogFile: file where ContextInterpreter stores context data.

applicationLogFile: file where user actions (e.g. start viewing, stop viewing, movie name etc are stored).

MetadataTreeFile, ContextTreeFile: same as required for SVM training, see above.

MetadataInfoFile: file which contains names of items and their metadata.

Examples of these files are also provided together with software.

ContextsToUse: context features which affect user choices in this application, same as in SVM training.

The function returns false if by some reason it can not start learning (e.g., it can not find required files). However, it is application responsibility to check that ContextInterpreter and all required context sources are running and log files are updated.

If no context data for analysis of implicit user feedback is available, it will be estimated only from viewed percent calculated from application log files, but this way is less intelligent.

The overview of this learning process and retrieval of user preference values (ranks) is presented in a figure below.

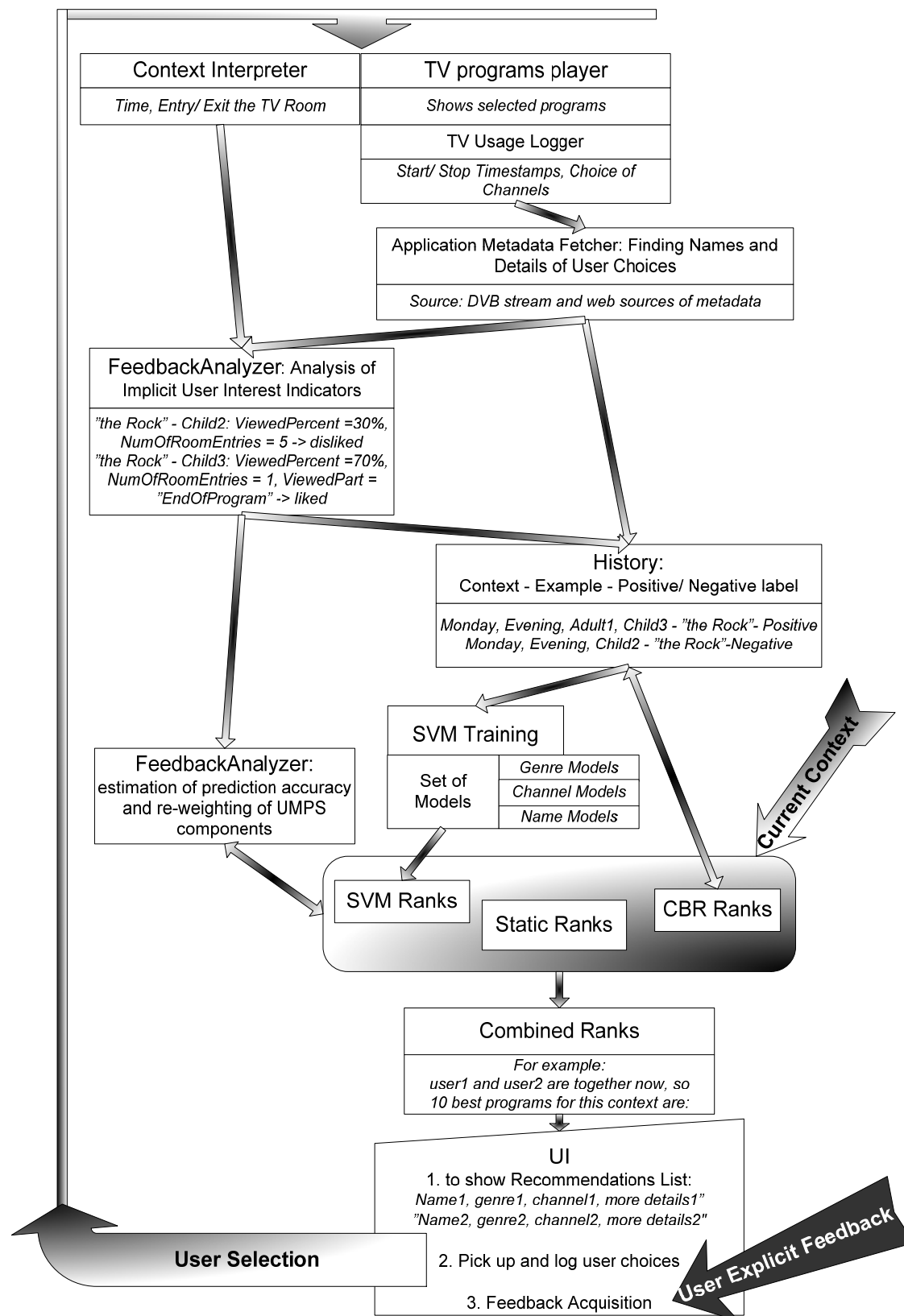


Figure 3.2: high-level overview of learning process with the examples from TV recommender application

3.2.7.4 Capability IModelMultimed:: getLearnedRanks(...)

string getLearnedRanks(string MetadataItem, string MetadataTreeFile, string CurrentContext)

Function parameters:

MetadataItem (same as SettingID)

MetadataTreeFile – same as required for SVM training

CurrentContext: list of context descriptors separated with whitespaces, e.g., “user1 user2

Sunday evening” (context includes also multi-user environments)

This function returns combined rank, calculated as weighted sum of ranks of three components of user model (static, CBR, SVM) where weights depend on past prediction accuracy of these components.

4 Tutorial

UMPS has different modules and interfaces offered to specialized groups of users:

- Stereotypes Manager: to be used by service and application providers in order to set the common sense knowledge of the system about its potential groups of users
- Service Interfaces: to be accessed by application providers at runtime to update user profiles and to get user preferences from different UMPS modules
- GUI for new profile generation: to be used by house system administrator to add new users to the installed system
- GUI for profile direct manipulation: to be used by system end-users to edit their own profile, and to add context-dependent profile settings

4.1 Install, configure and test functionality of compiled service components

The following instructions considers that you are going to use only the compiled service components, using the "UMPS.msi" (or older "UMPS.zip" archive) installation file that you received by e-mail or downloaded from gforge.

1. Run the msi installation file or unzip the archive "UMPS.zip" directly under "C:\"
2. What you should see after the install or the unzip (folders and files):

```
C:\UMPS\..\
    ..\config\..
        ..\ContextOntology.owl
        ..\log.txt
        ..\PersonalContextOntology.txt
        ..\PersonalContextOntology_initial.txt
        ..\SpecialContextTerms.txt
        ..\UMPSConfig.xml
        ..\UserOntology.owl
    ..\DB\..
        ..\UserProfile.mdb
        ..\UserProfileOntology.mdb
        ..\UserProfileVocabulary.owl
        ..\UserProfileVocabulary.pprj
    ..\Software\..
        ..\Client.exe
        ..\Constants.dll
        ..\CustomControls.dll
        ..\CustomTreeControls.dll
```

```
..\DataObjects.dll
..\DBInterface.dll
..\EMIC.FirewallHandler.DLL
..\EMIC.WebServerComponent.dll
..\EMIC.WSAddressing.DLL
..\FeedbackAnalyzer.exe
..\Functions.dll
..\Manager.exe
..\ReasoningModule.exe
..\StaticModeler.exe
..\UMPS_UI.exe
..\UserProfile.dll
..\WSDiscovery.Net.dll
..\UserProfiles\..
..\Jerry@JGAV300600.amigo.net.xml
..\Maria@JGAV300600.amigo.net.xml
```

3. Check functionality of services:

- a. Start the ReasoningModule service (double-click the file "C:\UMPS\Software\ReasoningModule.exe")
- b. Start the StaticModeler service (double-click the file "C:\UMPS\Software\StaticModeler.exe")
- c. Start the Client application (double-click the file "C:\UMPS\Software\Client.exe")
- d. On the Client console you should see a list of options to choose.
 - i. Choose "8" and press "Enter". On the Client console the userIDs of the two users existing in the DB should be displayed (Jerry@JGAV300600.amigo.net and Maria@JGAV300600.amigo.net) , as shown in figure 4.1.
 - ii. Choose "1" and press "Enter". Write to Client console "test" and press "Enter". A GUI should appear to let you give user details. Enter all compulsory fields in the GUI (yellow color) and press the "Add" button at the bottom of the form. You will be asked if you want to edit also context-dependent preferences. Press "No".
 - iii. Choose again "8" to the Client console. Now, except of the two previous users you should see the "test" userID also.
 - iv. Choose "2" to the Client console. Write to the console "text" and press "Enter".

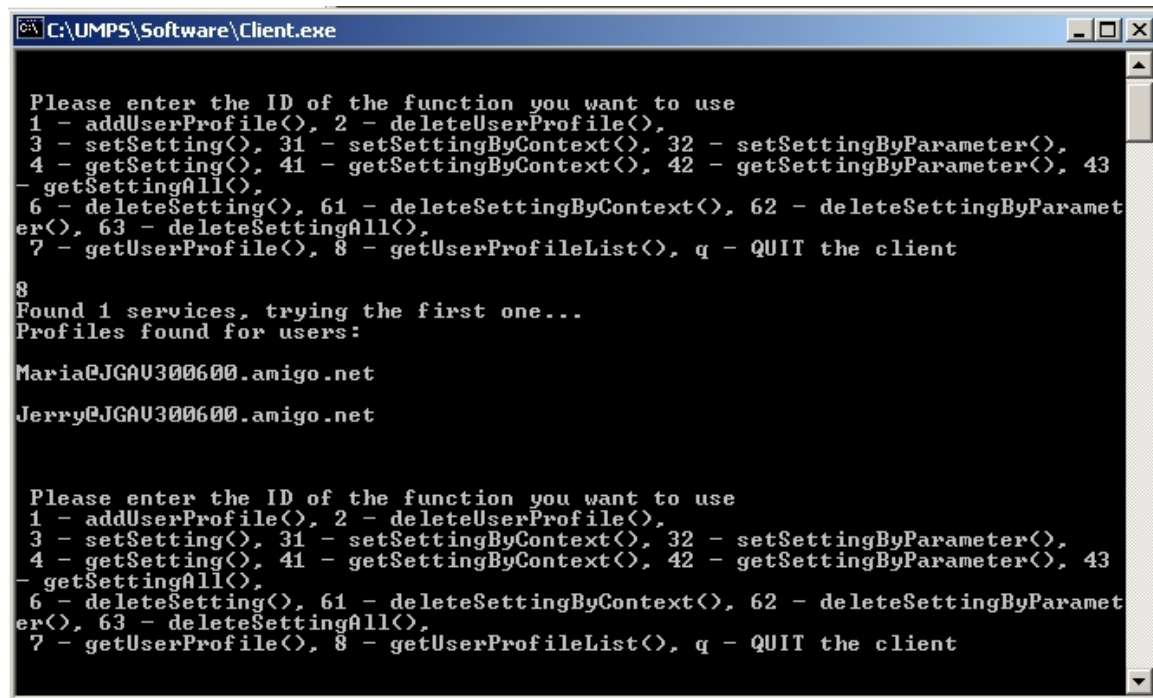


Figure 4.1. Instance of UMPS test Client interface.

4.2 Stereotypes Management

User profile initialization is based on a stereotypes library to overcome cost and time expenses of profiles initialization through the system.

Stereotypes library is configured and updated by system and application providers, not by end users!

Stereotype = settings, preferences and characteristics of a group of users.

Deeper position of the stereotype in the graph = smaller group of users and more accurate represented.

The following instructions suppose we want to add a new stereotype to describe a Muslim person.

1. Start "C:\UMPS\Software\Manager.exe" application, and the interface that allow you to edit the stereotypes database will be shown (see figure 4.2).
2. Browse Stereotypes library tree on the left to show different types of stereotypes.
3. Go to "RELIGIOUS_PERSON" stereotype. Right click with the mouse and choose "Add -> Stereotype". In the right side of the GUI, in the "Name" text box enter "MUSLIM". Press the "Apply" button on the bottom-right side of the GUI.
4. Right-click the "MUSLIM" stereotype and choose "Add -> Trigger". Enter trigger name "Muslim" and press "Apply".
5. Right-click the "MUSLIM" stereotype and choose "Add -> Class". In the right "Name" combo box choose "Preferences" and press "Apply".

6. Right-click the “Preferences” class under the “MUSLIM” stereotype and choose “Add -> Class”. Choose the “ProductsPrefs” name and press “Apply”.
7. Right-click the “ProductsPrefs” class under the “MUSLIM/Preferences” class and choose “Add-> Class”. Choose the “FoodPrefs” name and press “Apply”.
8. Right-click the “FoodPrefs” class under the “MUSLIM/Preferences/ProductsPrefs” and choose “Add -> Key”. Choose the “pork” name, enter Value “-5”, enter Rating “800” and press “Apply”. This has the meaning that “a Muslim person will most likely not eat pork meat”. The values should be from -5 (don’t like it at all) to +5 (like it very much). The Range of 800 means that the system is very confident on this information, as the scale is from 0 to 1000.
9. You can add any other keys that characterize this stereotype, as for example “Interests/Events/Ramadan” key, with Value = “5” and Rating “800”.

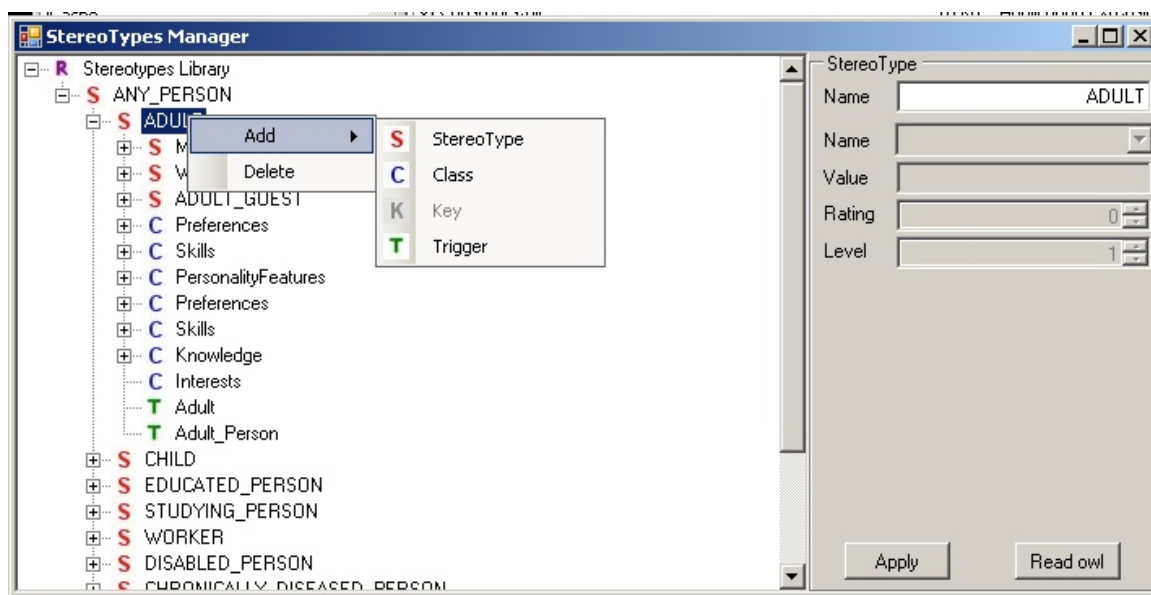


Figure 4.2. Stereotypes Manager interface.

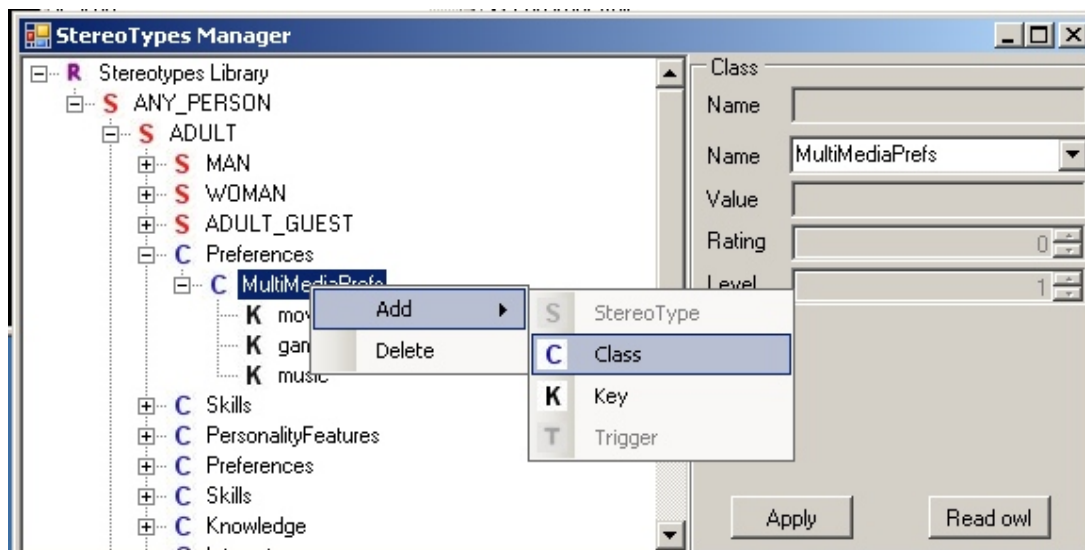


Figure 4.3. Addition of new classes or keys to already defined stereotypes.

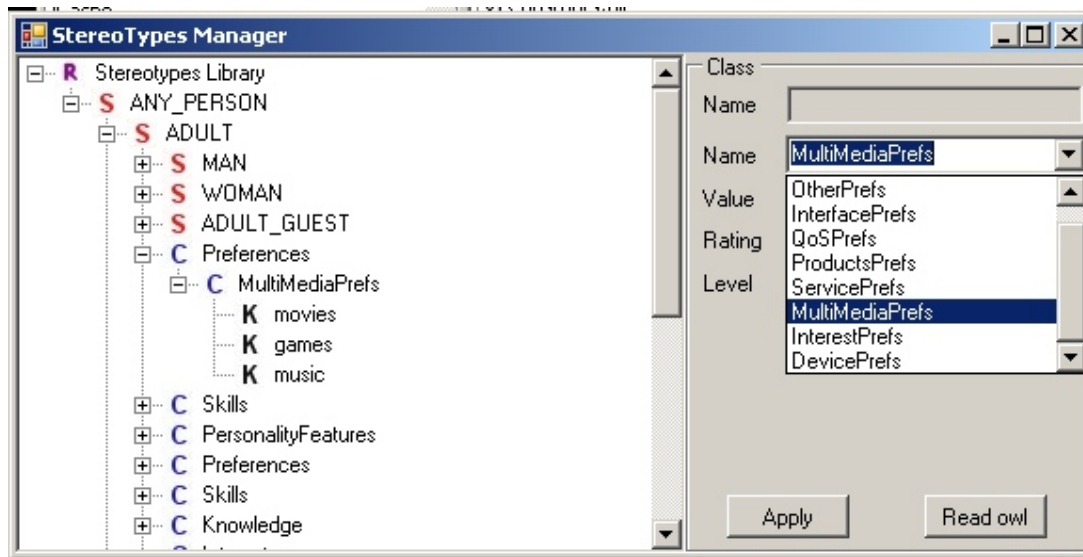


Figure 4.4. Class and key names are restricted to the ones defined in the user profile ontology.

4.3 Service Interfaces – how to be used by client applications

4.3.1 Use of static modeling functionality - `getSetting(...)`, `setSetting(...)` methods

Suppose the following scenario: you have an application ("MyApplication") which lets the user choose its own font size, and the application wants to keep this information for each device he uses in the house. Each device is described by a deviceID. The first time the application is run by the user, it will ask him which is his preferred font size (generally) and call the `setSetting(...)` to set this generally valid setting value:

```
setSetting("John@amigo.net", "Preferences:ApplicationPrefs:MyApplication:FontSize", "int", 12)
```

But the application also wants to keep this information for each device that John uses to run it. So, if John first run MyApplication from his bedroom PC ("JohnBedroomPC@amigo.net"), a first way to express that would be:

```
setSettingByContext("John@amigo.net",
"Preferences:ApplicationPrefs:MyApplication:FontSize", "int", 12, "context information describing the device ID")
```

or, if the application doesn't really care about well defined context description of the condition, the condition is very simple (has no timestamp or location stamp attached, or any other additional context), and this information is application specific (no other application or service in the system can make use of it) can define its own parameter to describe the condition:

```
setSettingByParameter("John@amigo.net",
"Preferences:ApplicationPrefs:MyApplication:FontSize", "int", 12, "deviceID",
"JohnBedroomPC@amigo.net")
```

After a while, John runs the application from its PDA ("JohnPDA@amigo.net").

The application checks John's profile to see which is his FontSize preference when running the application from its PDA, and will call either the `getSettingByContext(...)` or the

getSettingByParameter(...), depending on which one has been used to set the setting value. (I suppose is the second one):

```
getSettingByParameter("John@amigo.net",
"Preferences:ApplicationPrefs:MyApplication:FontSize", "deviceId", "JohnPDA@amigo.net")
```

If the returned value is NULL that means that the preference has not been set for the PDA. Thus the application will next call the

```
getSetting("John@amigo.net", "Preferences:ApplicationPrefs:MyApplication:FontSize")
```

to get the generally valid FontSize value and initialize with this FontSize the application on John's PDA.

But immediately after starting the application, John doesn't like the font size on its PDA, and change it to 8, so the application will call:

```
setSettingByParameter("John@amigo.net",
"Preferences:ApplicationPrefs:MyApplication:FontSize", "int", 8, "deviceId",
"JohnPDA@amigo.net")
```

to keep the preferred FontSize for John's PDA...

4.3.2 Use of dynamic modeling functionality - getContextBasedDynamicSuggestions(...), getFrequencyBasedDynamicSuggestions(...) and getNLastItems(...) methods

Suppose the following scenario: the users had set their preferences via GUI and started to use TV recommendation application, which stores user actions and the corresponding contexts in the log files. The context is stored as a set of descriptors, e.g., IDs of present users, day of week, time of day (semantic time, such as "morning" or "evening"), user events, weather etc. User actions are stored as names of viewed TV programs and movies, assuming that if the users viewed the program up to the end, they liked it to some degree. In order to recommend the TV program, UMPS can initially use only explicit user preferences, but users often can not describe (or are lazy to describe) their preferences with sufficient precision. So when large enough application log files are collected, UMPS can retrieve recommendations also based purely on dynamic modeling functionality. For example, from log files it can be concluded with high confidence that users often watch some program on Friday evening, although the users did not set high preference value for genre of this program via GUI. The getContextBasedDynamicSuggestions function of Dynamic Modeller will recommend this TV program for Friday evening, and if application designer considers that the recommendation is given with high enough confidence, it can be added to the list of recommended items. However, what is "high enough confidence" is very much application dependent. For example, if failure to recommend some item is more dangerous for trust in application than recommending extra items (which is the case for applications, simply recording TV programs for future use), the items should be recommended even if the confidence is fairly low. For applications which recommend items for immediate use (for a example, applications which decide whether to remind a user to do something or not) the confidence value should be very high.

Another example of use of Dynamic Modeller functionality is the following: suppose that some movie gets high ranks because the user has set high preference values for the movie genre and actors. However, if getNLastItems function of Dynamic Modeller returns this movie among the last 5 viewed movies, perhaps the rank of the movie should be lowered because people don't normally view the same movie two times in a row. Use of getNLastItems function is very much domain-dependent, however. For example, if last 10 times the user was retrieving weather forecast from some particular web page, most probably it means higher trust in this weather station, and the corresponding web page should be ranked highest.

Use of function `getFrequencyBasedDynamicSuggestions` is also domain-dependent, for example, if most frequently viewed sports program is downhill skiing, this can mean that preference values of all family members for this program are context-independent: all family members watch downhill skiing competitions together, separately, on weekdays and weekends, in the mornings and in the evenings and so on. On the other hand, if most frequent food on the table during last week was pizza, perhaps some other food product should be recommended for the next week menu, because eating just pizza is not healthy.

Since use of recommendations, provided by Dynamic Modeller, is very much domain-dependent, UMPS does not provide a way to combine outputs of its different components. Applications can decide, how to use which UMPS components in the most suitable way.

4.3.3 Use of Multi-Profile aggregation functionality

`getMultiProfileContextBasedDynamicSuggestions(...)`

`getMultiProfileFrequencyBasedDynamicSuggestions(...)`

`getMultiProfileNLastItems(...)` methods

Multi-Profile aggregation of preferences of multiple users from application log files allows avoiding reasoning how people in a group would resolve conflicts between their interests. Instead, the aggregation of preferences is done by observing how users resolve these conflicts: items are ranked based on how many users from a group were present when the items were retrieved.

The first two functions will recommend some TV program if it was viewed by the group of users when they were gathered together. However, these two functions are different. The latter function counts only cases when all users from a group were present, and does not consider other context descriptors (such as day of week). The former function ranks highest the items which were retrieved when all users from a group were present, and were not retrieved when some of group members were absent. Among these items ranks slightly differ, depending on the similarity of the current context with the contexts stored in a history, for example, if today is Friday, items retrieved on Friday will be ranked a bit higher than items retrieved on other days. The items which were retrieved when all group members were present, as well as when some group members were absent, get lower ranks, and the items which were never retrieved when all group members were present, are ranked much lower. Choice between these two items is application-dependent, for example, if one of group members has a food allergy, an application should recommend only food items retrieved when all group members were present. `GetMultiProfileFrequencyBasedDynamicSuggestions` suits better to this goal. However, if an application should recommend music for some party, it is not so important to satisfy all present guests, because some guests might never go dancing. However, it is important to recommend dancing music, thus context type "party" must be considered. Thus, `GetMultiProfileContextBasedDynamicSuggestions` function suits better.

Same as in case of Dynamic Modeller functionality, use of `getMultiProfileNLastItems` function is very much domain-dependent: the same movie should not be recommended several times in a row, whereas the series of the same soap opera or the same channel of TV news should be recommended.

Apart from aggregation of multi-user preferences from application log files, the Multi-Profile Aggregator performs aggregation of explicitly acquired user preferences. Two ways to aggregate preferences are provided: to get union of preferences (e.g., a shopping application should buy different chocolates for different family members plus beer for a father) and intersection of preferences (e.g., a cooking application should not select for a family dinner a food recipe which some family members will not eat).

According to study of how humans solve the task of profile aggregation (see J. Masthoff, Group Modeling: Selecting a Sequence of Television Items to Suit a Group of Viewers, User

Modeling and User-Adapted Interaction 14, 2004, 37-85), most common strategies are the following:

1. Average strategy: a weighted sum of users' preference values. Weights can be assigned by an application developer, e.g., preferences of a guest can be very important.

Weights can be learned from the application log files, e.g., it can be observed that father's preferences often dominate in the evening when father comes from work. This is the most democratic strategy, which allows to take into account opinions of all users.

2. Least Misery strategy: minimum of users' preference values. This strategy is based on the belief that a group of users is as happy as the least happy group member, and an item should be recommended only if all group members like it. This strategy is most suitable, if the task is to find an item which all users will certainly like.

3. Average Without Misery strategy: a weighted sum of users' preference values, if preferences of ALL users exceed certain threshold. If preference value of some user is less than the threshold, the item is treated according to the minimum of users' preference values. This strategy is the best if the task is to find an item which all users will not certainly dislike, and to rank highest an item which most users will like.

UMPS Multi-Profile Aggregator implements all three popular strategies, and application designers can decide which one suits better to the task at hand.

Since use of recommendations, provided by Multi-Profile Aggregator, is very much domain-dependent, UMPS does not provide a way to combine outputs of its different components. Applications can decide, how to use which UMPS components in the most suitable way.

4.4 Use of SVM

SVM (Support Vector Machines) is a neural network initially developed for two-class classification problems (an example of typical SVM use is biometric verification when a person can be either "genuine" or "impostor", and nothing else). However, SVM is also frequently used for multi-class classification problems because it usually shows good generalization capabilities.

We used SVM implementation in TORCH machine-learning library in this project. SVM is very nice in a sense that it does not require manual selection of its parameters (unlike neural networks which require application developer to choose number of nodes). Parameters of SVM can be also tuned, but usually it does not affect prediction accuracy significantly. Thus, any application developer can use SVM with its default parameters and get useful predictions if application provides large number of training data and if features of data were carefully selected, that is, if most of features help separation of data into two classes and none of features adds too much noise. For example, in our tests with learning TV program preferences we observed that social context (who of family members is present) was useful feature for most of families (although in some cases time context-based predictions were more accurate), while day of week in some families was useful feature, and in some families prediction accuracy was better if we only distinguished between "workday", "Friday" and "weekend" days of week.

So general advice which we can give is the following: do not train SVM on data which contains less than 10 positive examples of user choices; and carefully select context features which affect user choices in each particular application. For example, shopping lists are usually affected by approach of public holidays, trips and parties, while for TV program recommender system such context types would mainly add noise to the training data. On the other hand, shopping lists are not often affected by the "day of week" context type, while for TV program recommender system this context type is very useful.

4.5 Use of FeedbackAnalyzer and Context-Dependent Learning functionality

As it is already said before, UMPS is a modular system, and applications can get full control over use of its components (most of UMPS components can be used independently from each other). This way we ensure that UMPS can be used by many applications in a flexible way, because we observed that usefulness of different components is very much application-dependent and domain-dependent. For example, FeedbackAnalyzer can be used for evaluation of user feedback in many ways because it is quite generic reasoning component, and applications can create different rules with it according to own logic, availability of context sensors and distribution of sensors over environment.

However, we also developed and tested (on real life data of viewing TV) automatic learning functionality of UMPS by joint work of several UMPS components. This way application is required to provide some configuration inputs, to start context acquisition and log, to start application log and to wait until history of user actions in different contexts becomes sufficiently large for making useful predictions (which might never happen if user behavior is more or less random or if context features were poorly selected). Since overall UMPS rank is a weighted sum of ranks of three components: Static Model, CBR and SVM, in case of poor recommendation accuracy of both CBR and SVM UMPS will mainly use ranks of Static Model; however, this does not guarantee good prediction accuracy as well because users can set preferences in a way unpredicted by system designers (e.g., if users' opinion of what is "action movie" differs from understanding of application developer). Examples of all required files are provided together with software, so application developers can start learning on provided files and observe how it goes.

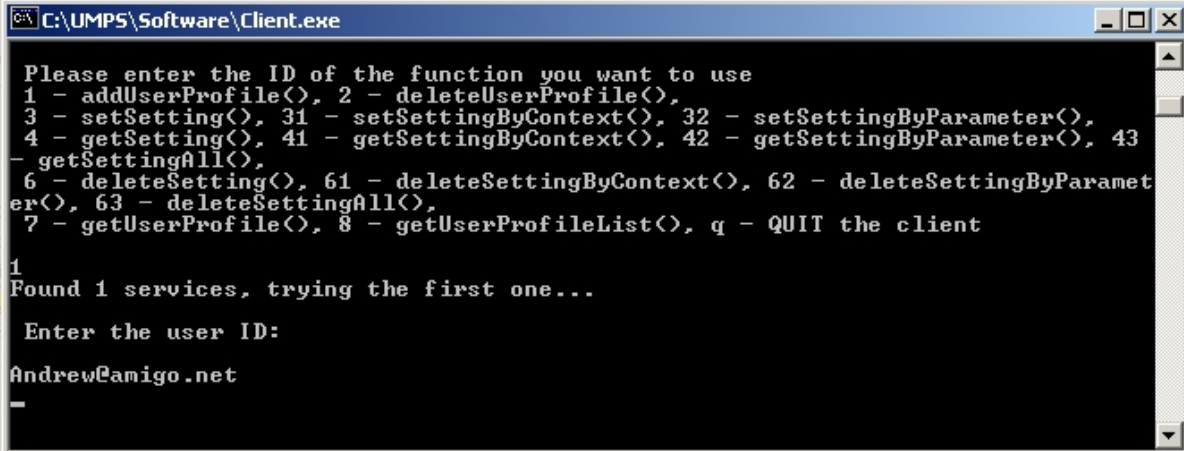
Generally, we suggest that application developers should remember that humans nature is largely unpredictable and current state-of-the-art user modeling methods achieve good recommendation accuracy with real life data in approximately 70% of cases, thus leaving out one third of user choices – pretty much. For example, collaborative filtering is considered a good recommendation method, although it requires a lot of manual user effort for ranking items and has privacy implications (requires data sharing between users), that's why we did not use collaborative filtering in Amigo. Two state-of-the-art works present prediction accuracy of two collaborative filtering-based systems: the work (G. Adomavicius, Y. Kwon, New Recommendation Techniques for Multicriteria Rating Systems, IEEE Intelligent Systems, Vol. 22, No. 3, May/June 2007, pp. 48-55) reported a system with precision in top five recommendations by different methods ranging from 65% to 75%, while precision of random recommendations was only twice lower (35.6%). Another work (B. Smyth, P. Cotter, A personalized television listings service, Communications of the ACM 43(8), 2000, pp. 107-111) reported a system which was evaluated as "good" by 61% of users, and it was a successful result. Results of UMPS tests are presented in Appendix 5.5.

4.6 GUI for new profile generation

Personal details are directly acquired from the user, and added to the profile. These data are also analyzed to extract triggers and activate stereotypes (i.e. if age <18 trigger="Child", else trigger="Adult", if Religion="Muslim" trigger="Muslim", etc.). The following instructions show how to add a new user profile, for Andrew, a family friend.

1. Start "C:\UMPS\Software\StaticModeler.exe" service.
2. Start "C:\UMPS\Software\ReasoningModule.exe" service.
3. Start "C:\UMPS\Software\Client.exe" application.
4. Choose option "1" and press "Enter".

5. Enter to the Client console the userID for the new user profile ("Andrew@TMCV110203.amigo.net") and press "Enter"
6. The GUI for specification of personal details will appear.
7. Set all the compulsory fields in the "Personal Details" tab, and don't forget to set an "adult" date of birth – else Andrew will be considered a child.
8. Go to the "Additional Personal Details" tab, enable the Religion and choose "Muslim".
9. Go to "Security & Privacy" and choose for User Type "Guest".
10. Press the "Add" button at the bottom of the form.
11. You will be asked if you want to edit also context-dependent settings. Select "Yes". A new GUI will be displayed, initialized with the profile data of "Andrew@...".
12. Browse the profile tree and go to "Preferences/ProductsPrefs/FoodPrefs" to see the "-5" value for "pork" coming from the "MUSLIM" stereotype...



```
C:\UMPS\Software\Client.exe

Please enter the ID of the function you want to use
1 - addUserProfile(), 2 - deleteUserProfile(),
3 - setSetting(), 31 - setSettingByContext(), 32 - setSettingByParameter(),
4 - getSetting(), 41 - getSettingByContext(), 42 - getSettingByParameter(), 43
- getSettingAll(),
6 - deleteSetting(), 61 - deleteSettingByContext(), 62 - deleteSettingByParameter(),
63 - deleteSettingAll(),
7 - getUserProfile(), 8 - getUserProfileList(), q - QUIT the client

1
Found 1 services, trying the first one...

Enter the user ID:
Andrew@amigo.net
-
```

Figure 4.5. Use of UMPS test Client application to add a new user profile.

The screenshot shows a window titled "UserProfile for Andrew@amigo.net" with a standard Windows XP-style title bar. Below the title bar is a tabbed interface with five tabs: "Personal Details", "Additional Personal Details", "Products", "Security & Privacy", and "Multimedia". The "Personal Details" tab is currently selected. The form contains the following fields:

Field Label	Value / Selection
Last Name	[Redacted]
First Name	Andrew
Middle Name	[Redacted]
Gender	Male
Date of Birth	2/27/1970
Country	Netherlands
City	Amsterdam
Address	[Redacted]
Occupation	Working_Person
Profession	Manager
School Level	[Redacted]
Education Level	High_Educated_Person

At the bottom of the window, there are three buttons: "Add", a button with a red 'X' icon, and "Close".

Figure 4.6. New User Profile Generation interface.

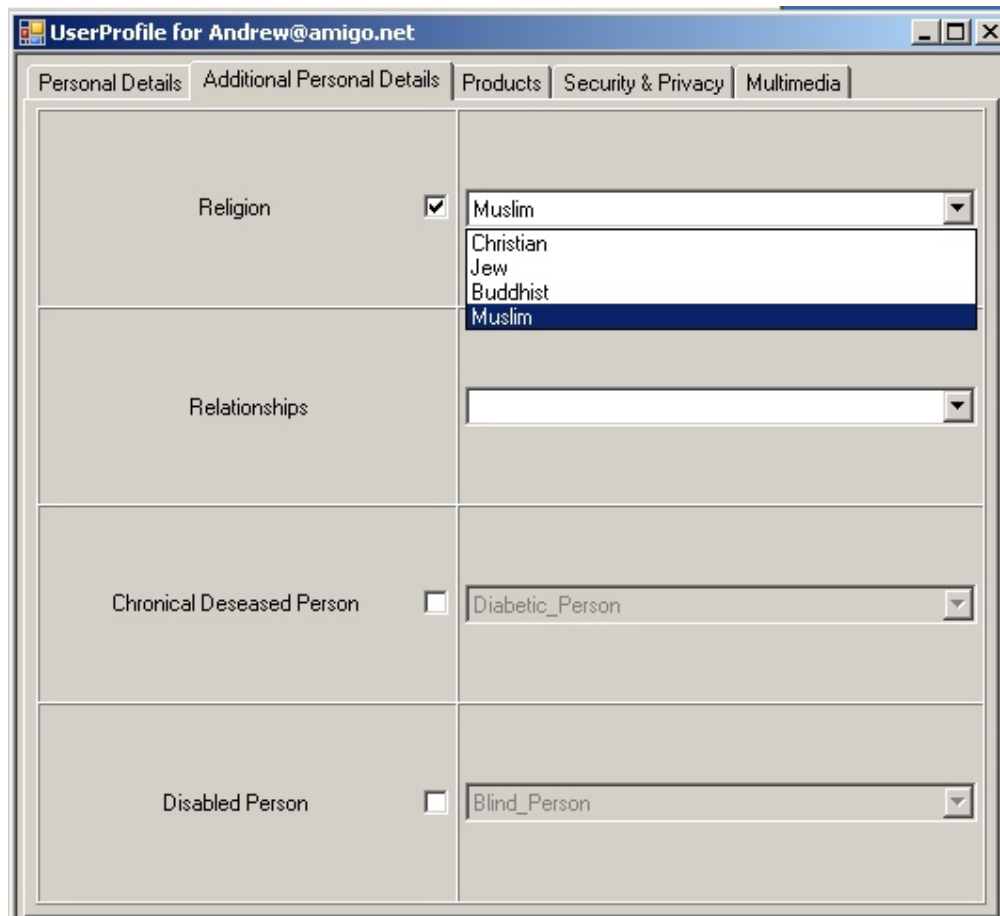


Figure 4.7. Selection of Additional Personal Details.

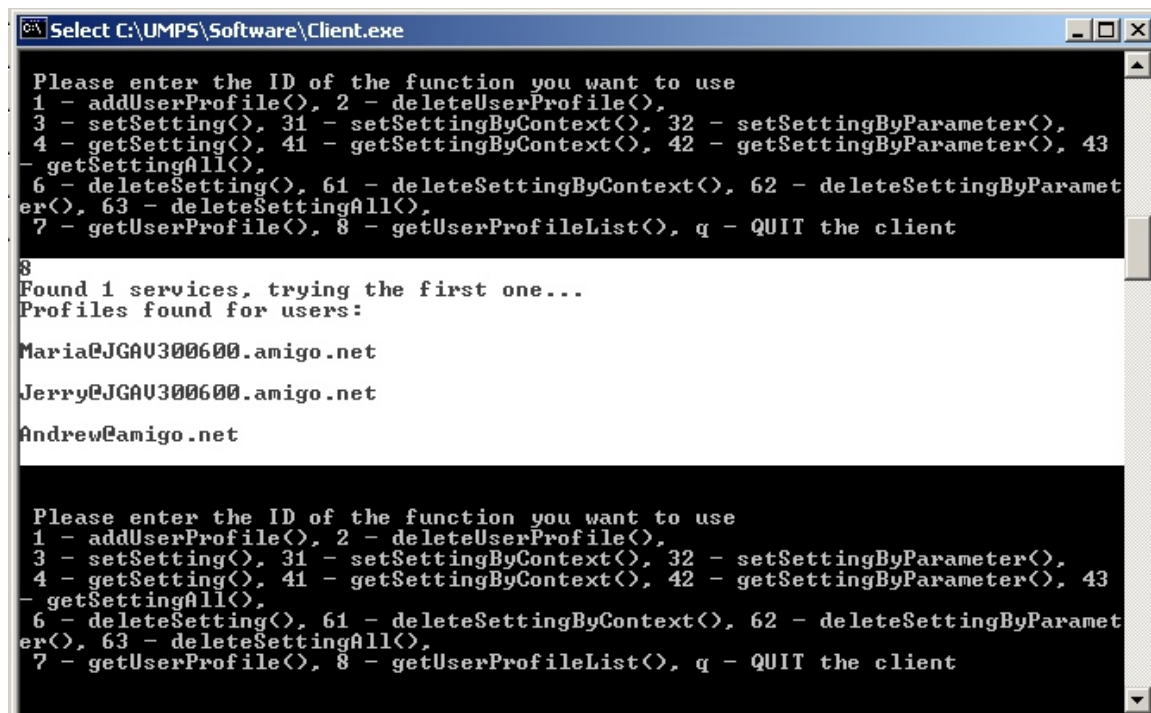


Figure 4.8. Use of the UMPS test Client to verify the addition of a new profile

```

Select C:\UMPS\Software\Client.exe
3 - setSetting(), 31 - setSettingByContext(), 32 - setSettingByParameter(),
4 - getSetting(), 41 - getSettingByContext(), 42 - getSettingByParameter(), 43
- getSettingAll(),
6 - deleteSetting(), 61 - deleteSettingByContext(), 62 - deleteSettingByParamet
er(), 63 - deleteSettingAll(),
7 - getUserProfile(), 8 - getUserProfileList(), q - QUIT the client

4
Found 1 services, trying the first one...

Enter the user ID:
Andrew@amigo.net

Enter the settingID:
PersonalDetails:FirstName
The query result is:

<key id="FirstName"><valueset><value type="string">Andrew</value><rating>1000</r
ating><justification>Explicit</justification></valueset></key>

Please enter the ID of the function you want to use
1 - addUserProfile(), 2 - deleteUserProfile(),
3 - setSetting(), 31 - setSettingByContext(), 32 - setSettingByParameter(),
4 - getSetting(), 41 - getSettingByContext(), 42 - getSettingByParameter(), 43
- getSettingAll(),
6 - deleteSetting(), 61 - deleteSettingByContext(), 62 - deleteSettingByParamet
er(), 63 - deleteSettingAll(),
7 - getUserProfile(), 8 - getUserProfileList(), q - QUIT the client

```

Figure 4.9. Use of the UMPS test Client to request data (setting values) for an existing user profile.

4.7 Direct Manipulation GUI tutorial

This interface provides the way for the users to view and to manually set/ change their preference values that are already stored in the profile. The GUI also provides to application developers a comfortable way to simulate user profile data for application developers, e.g. needed for testing an application.

The GUI works in a following way: tree of user preferences is read from User Modeling Ontology file and presented via GUI with default preference value equal to the lowest limit of user preference (which indicates that the user has no interest in this topic). The user can change any preference value, and to attach to it any context from Context Ontology.

Similar, the tree of user personal details is presented, so the user can change Personal Details. More details about UI below:

Before editing user profile, a user is required to enter ID (later we will change it for entering user name and a password).

Figure 4.10 presents the start page. List of already existing IDs is presented, so a new user can just enter a number which is not present in the list. The user ID should be entered in the field below "User ID" text.



Figure 4.10. Start page of the UI

After the user enters his ID, his preferences are loaded, and the user sees the next page. The leftmost box shows user profile tree, the rightmost box shows the tree of contexts which the system is able to recognize.

The user can click on a "+" sign, and the tree will be expanded. The user preference values are attached only to the leaf nodes of the tree ("leaf node" means a node which can not be expanded anymore). The default value for user preferences is 0. If the user preference is unknown, it's value is shown as 0. For personal details the default value is an empty string.

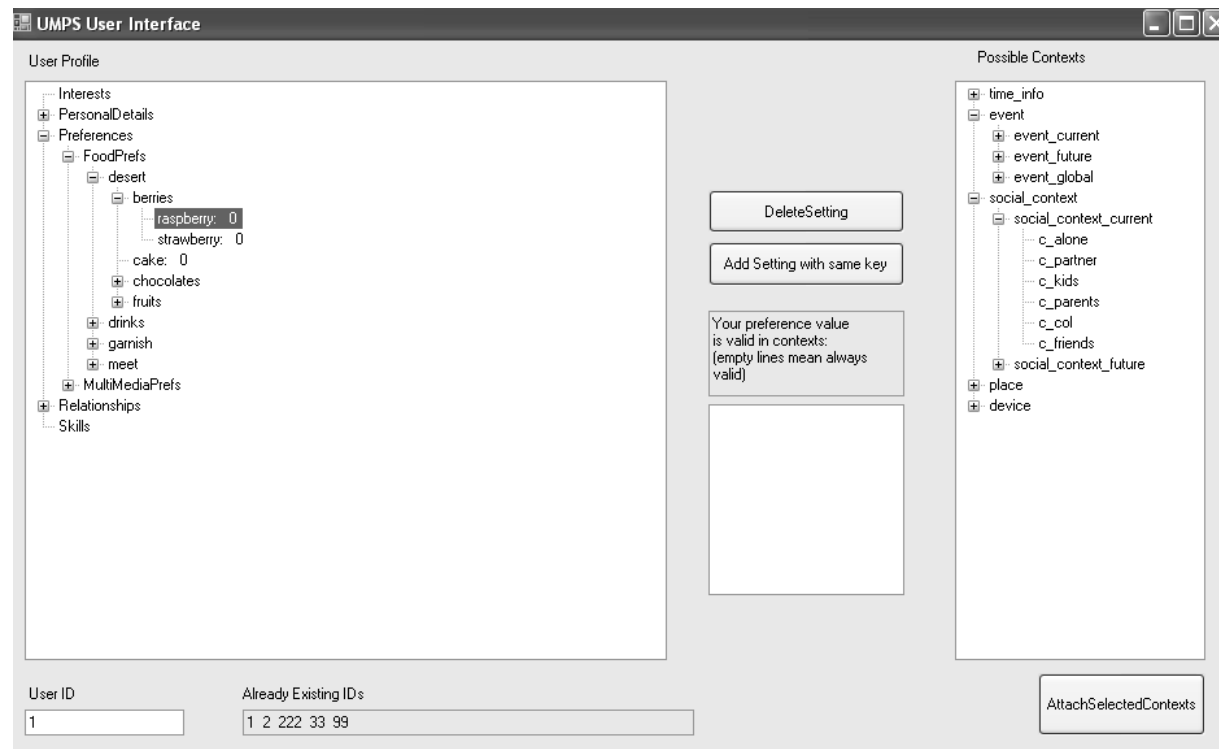


Figure 4.11. Next page of user profile editing for a new user (all user preferences are set to 0, and no contexts are attached).

The user can change a value of any leaf node. The user needs to select a node and double click with the mouse. After that a box with the prompt “Enter value here” appears and shows the current preference value (see Figure 4.12). For numeric values of user preferences the prompt also informs the user what are the upper and lower limits of preference values.

When the user enters a new value and presses “Accept Change” button, the new value is written to the user profile, and appears in the leftmost window. However, if the user enters the numeric value above the upper limit or below the lower limit, the change is not accepted. After the user confirms the change of the preference value, the text box “enter new value” and the button “Accept Change” disappear, so if the user wants to change another value, he needs to double click on it again.

The user can attach contexts to the value by selecting a leaf node in a context tree and pressing “Attach Selected Context” button. The contexts attached are shown in the profile tree and in the window under the caption “Your preference value is valid in context(s):” (see Figure 4.12).

If there is no context attached, the value is assumed to be always valid.

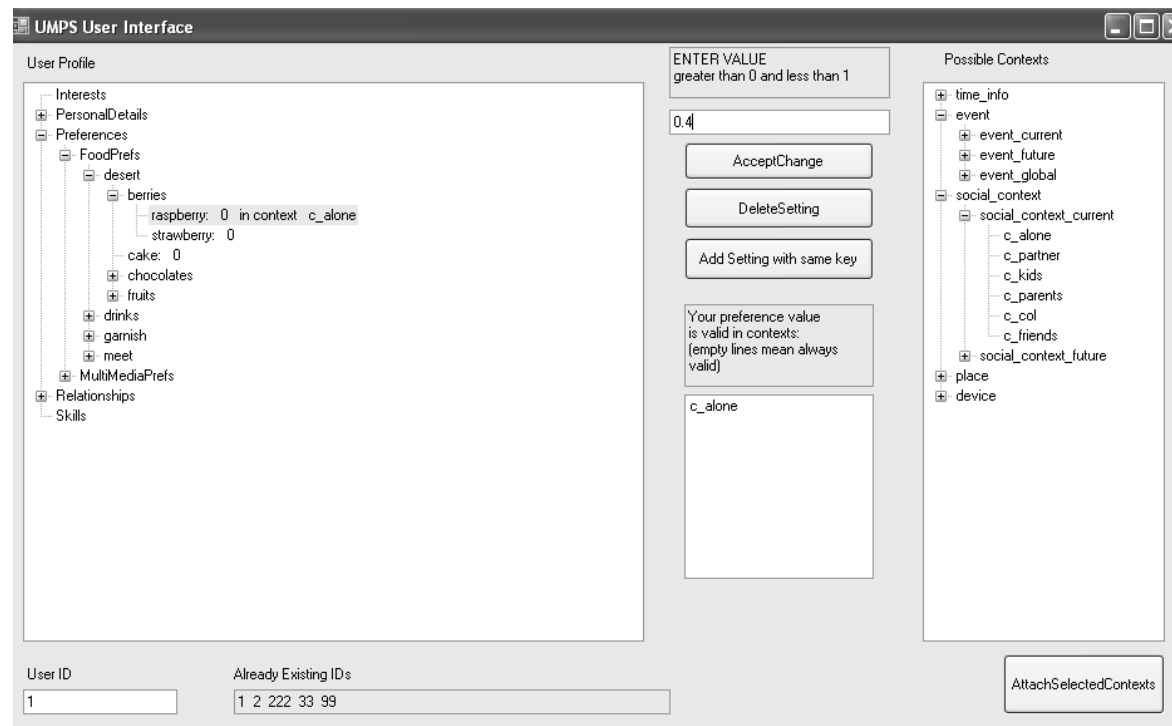


Figure 4.12. How changing values and attaching contexts works

The buttons “Delete Setting” and “Add Setting with the same key” allow the user to delete any leaf node in the user profile tree, and to duplicate any leaf node.

When the user presses “Add Setting with the same key” button, the leaf node with the same key and with the default value is added to the tree. This is needed in order the user can have different preference values for different contexts, see Figure 4.13.

The button “Add Setting with the same key” is needed also for entering personal details, e.g., the user can enter as many nicknames, names of children or friends as he wants.

When the user presses “Delete Setting” button, the selected leaf node is removed from the tree.

The screenshot shows the UMPS User Interface. On the left, the 'User Profile' section is expanded, showing 'PersonalDetails' and 'Preferences'. Under 'PersonalDetails', fields like Address, City, Country, FirstName (Thomas), Gender, Height, LastName, MiddleName, Nationality, NickName (Tommy), and NickName (Tom) are visible. Under 'Preferences', 'FoodPrefs' is expanded, showing 'desert' with 'berries' (raspberry: 0.4 in context c_alone, raspberry: 0.9 in context c_kids, strawberry: 0) and 'cake: 0'. Other food preferences like chocolates, fruits, drinks, garnish, and meat are also listed. On the right, the 'Possible Contexts' section shows a tree structure with 'time_info', 'event' (event_current, event_future, event_global), 'social_context' (social_context_current with sub-contexts c_alone, c_partner, c_kids, c_parents, c_col, c_friends; and social_context_future), 'place', and 'device'. Below the 'Possible Contexts' list is an 'AttachSelectedContexts' button. In the center, there is a text input field with '0.9', buttons for 'AcceptChange', 'DeleteSetting', and 'Add Setting with same key', and a box titled 'Your preference value is valid in contexts: (empty lines mean always valid)' containing 'c_kids'. At the bottom, there is a 'User ID' field with '1' and an 'Already Existing IDs' field with '1 2 222 33 99'.

Figure 4.13. How the user can have different settings for different contexts, and multiple entries in Personal Details

If the user wants to remove some of contexts attached to the preference value, he can do it by selecting the context in box under the caption “Your preference value is valid in contexts” and by pressing “Remove Context” button (see Figure 4.14)

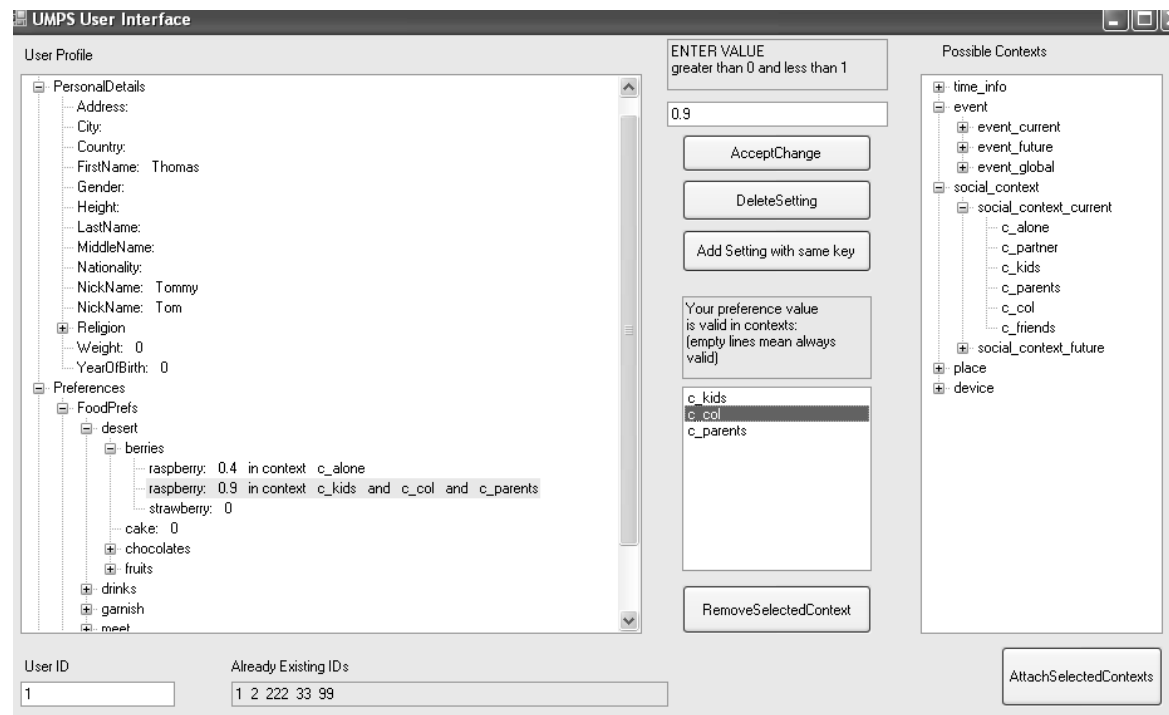


Figure 4.14. How the user can remove erroneously attached contexts

Generally there are two ways of explicit acquisition of user preferences: system-initiated and user-initiated. Most of current systems employ system-initiated way (the list of metadata items and order of profile acquisition steps are defined by application developer, and the users have to follow this procedure), which has a disadvantage that users have to look through long lists of actors' names or movies' names in order to find a few favorite actors and to assign high preference values to these actors' names. Furthermore, with system controlled profile acquisition users might be never able to find a favorite actor in the list if ontology developers decided against including this actor into the list or were simply unaware of his existence.

Experiments with user-controlled profile acquisition in MovieLens have shown that user-controlled profile acquisition leads to an equally good or even better user model, and additionally increases user satisfaction. However, user-controlled profile acquisition is rarely used because of a danger that users forget some important setting or mistype it. In this project we decided to use mixed approach: application developers provide an ontology which includes most important terms from their point of view, and users can add most important terms under existing terms. The responsibility to check grammar of new terms is with the users; however, we think that users will only add personally most important metadata items and thus it is quite probable that the users will either remember how to write them correctly, or will take an effort to check the spelling with Google. Figure 4.15 shows part of GUI where the user adds favorite actor to the ontology.

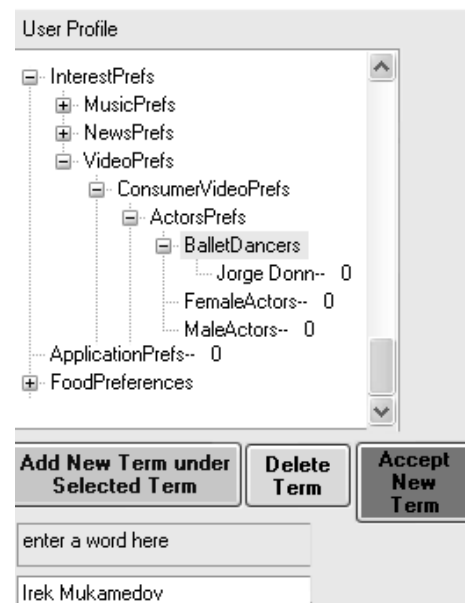


Figure 4.15 Part of GUI which shows how the user can add a personally important metadata term

We found it very important to allow users also to add personal context terms into context ontology. Humans think in terms “Jim’s birthday”, “Bridge evening” rather than in exact hours, dates and lists of game players. Since such high-level semantic contexts are personal, we allow users also to add personal context terms into context ontology under existing concepts provided by application developers. For example, “Public holiday” context is important for many applications, but public holidays vary in different countries, and users’ preferences may depend or may not depend on what kind of public holiday it is. For example, a person living in Finland can add “Independence Day” concept under “Public Holiday” if this person has special preferences for Independence Day (e.g., wants to watch reception in president palace by TV)

In order to help the users to map user-defined concepts into machine-readable terms, we require application designer to specify sets of important sub-concepts for each ontology term, e.g., for a “Public Holiday” concept the most important things to know are: whether its date is fixed or not; exact or approximate dates; country and duration. This set of sub-concepts is presented to a user at once for editing, and then Context Interpreter component of CMS can use this set for recognition of personal contexts if needed.

This approach has an additional advantage of helping in processing calendar events. During our system development we found that if users think about some regular event in terms of e.g. “photo club party” or “camp fire evening”, in most cases they would add the same term via GUI if they want to specify preferences for such personal contexts. Figure 4.16 (top) shows how the user adds personal context term “hki” (same term as he usually uses when he writes his schedule in his mobile phone) in order to specify preferences for this personally important frequent event, and how he edits the details of this term in the text box, so the system knows the location of this personal work trip (city of Helsinki), and other details. Lower part of figure 4.16 shows how the user specifies preferences for this context.

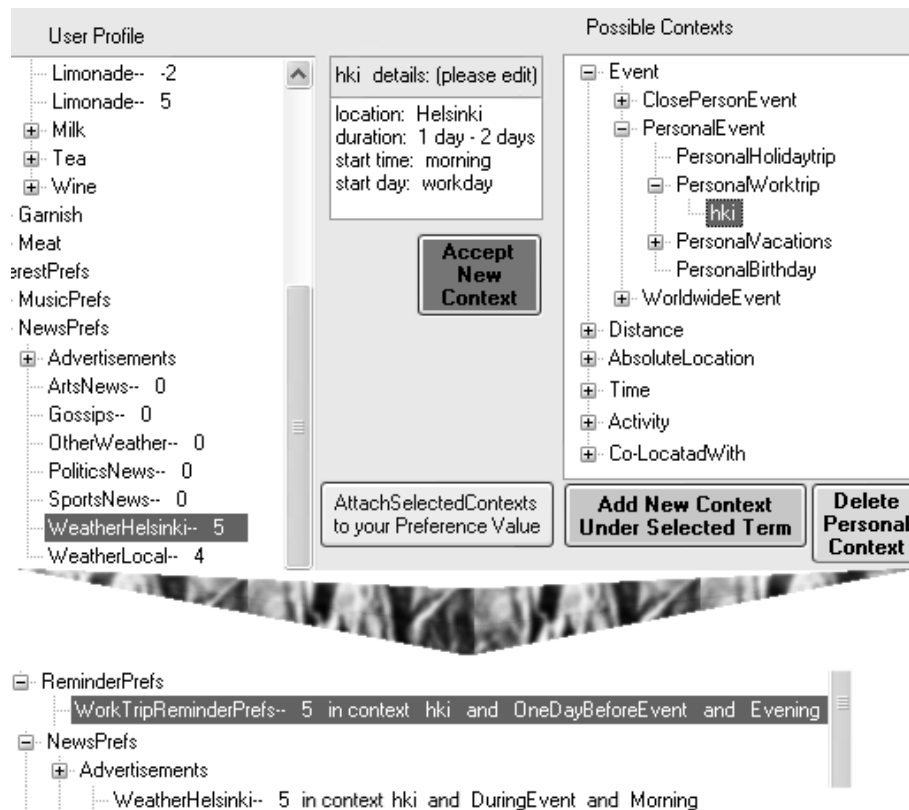


Figure 4.16 Top: part of GUI which shows how the user can add a personally important context term and provide its details. Bottom: personal preferences set for this personal context.

Allowing users to add personal context terms and personal metadata terms could require that each user has three personal ontologies: personal ontology of preference terms, personal ontology of contexts and personal ontology of mapping from custom context terms to context terms provided by application designer. First we tried this approach, but then we observed that since family members depend on events in lives of other family members, it is useful to have a family ontology of contexts instead of having own context ontology for each user. We also observed that it is useful to have a family ontology of preference terms because it reminds users to set negative preference values to favorite actors of other family members, if the users want so, and thus to achieve better control over recommendations.

5 Appendix

5.1 User profile ontology

Extension of Amigo ontology was needed by UMPS, and such ontology was defined in alignment to the already defined Amigo ontology in WP3. Most of this ontology has now been integrated in the Amigo ontology. This ontology can be found in the `../UMPS/DB/UserProfileVocabulary.owl` file. There is also an Protege 3.2 project file (`../UMPS/DB/UserProfileVocabulary.pprj`) which can be used to update the user profile ontology used. In this case the update of the `UserProfileOntology.mdb` should be consider, by using the `Manager.exe` application.

5.2 Stereotypes

A stereotype is a collection of preferences that can describe a certain group of users of the system, and may range from very general to very specific ones (i.e. adult, male, medical doctor, catholic). The stereotypes are arranged into a directed acyclic graph (see the example presented in figure 5.1) formed by the partial ordering relation “generalization of.”, allowing information not to have to be represented identically in many different stereotypes. The most general node of any stereotype structure is the stereotype `ANY_PERSON`.

Each preference in the stereotype represents a “key”, and has associated a “value”, and each pair “key”-“value(s)” has associated a rating. The rating is a representation of how confident is the system that the “key”-“value” pair to which is associated is correct for that group of users. In addition, the potential use of associated context to each key will be investigated.

The stereotypes can be considered as the common sense knowledge of the system about its potential groups of users, representing valid assumptions normally true in normal conditions [Ric98].

Users characteristics (keys) representation and grouping in the stereotypes are based on user ontology definition in the system (deliverable D3.1a), with some changes as presented in section 3.2.4. Stereotype keys are grouped in the following categories: *Preferences*, *Interests*, *Knowledge*, and *Skills*. The subclasses of *Preferences* used in stereotypes are: *InterfacePrefs*, *MultiMediaPrefs*, *ProductsPrefs* and *OtherPrefs*. One subclass of *Interests* present in stereotypes is the *Events* class. *MultiMediaPrefs* are further classified in *MoviePrefs*, *MusicPrefs*, *NewsPrefs*, *GamesPrefs*. For each of these last subclasses, a wide variety of subclasses is present, however only some of these being used in stereotypes, such as: *MovieGenrePrefs*, *CountryOfOriginPrefs*, *PoliticsNewsPrefs*, *SportsNewsPrefs*, *WeatherNewsPrefs*, *GossipsNewsPrefs*, *ShoppingAdsPrefs*, *GamesGenrePrefs*, *MusicGenrePrefs*, etc.

The stereotypes used are based on the categorization of user types in the system. These stereotypes will form the library to be used at system initialization, making possible to avoid the costly and elaborate construction of user ontology through the adaptive system itself. The initiation process for adaptive systems is thereby greatly simplified.

STEREOTYPE Hierarchy: Direct Acyclic Graph

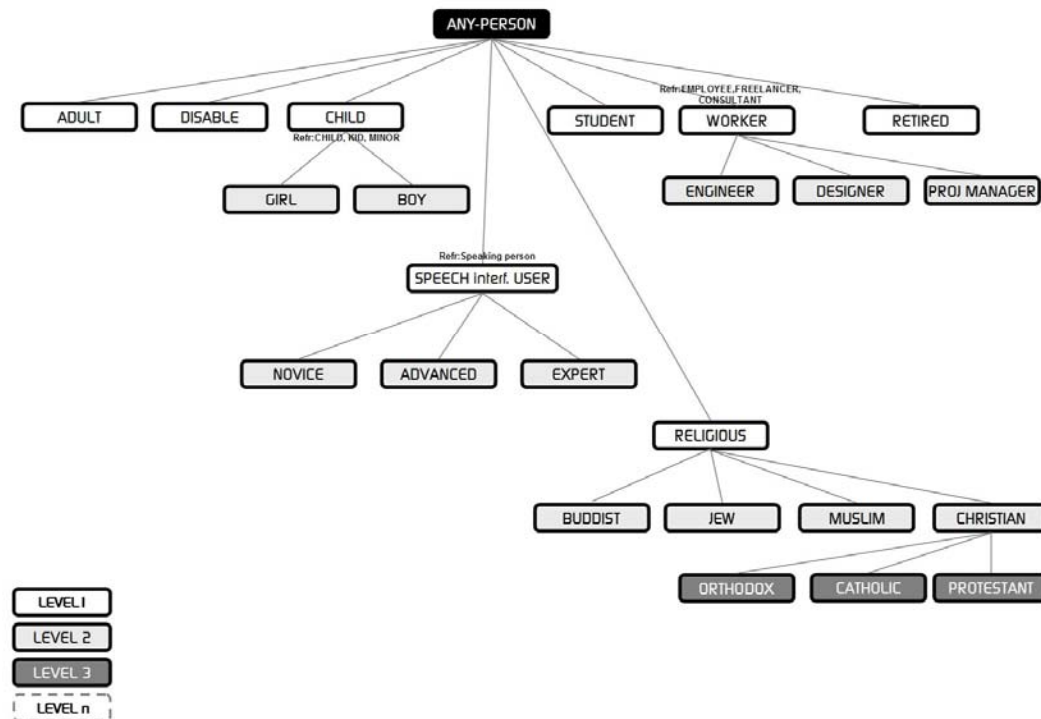


Figure 5.1: Stereotypes direct acyclic graph

5.3 User profile

The user profile is a tree-based representation of individual user preferences and personal data. Preferences and data are grouped, in agreement with user ontology representation in the system, in a similar way to user characteristics representation and grouping in stereotypes.

Initially, the profile is built by combining the preferences from activated stereotypes with explicitly acquired preferences and personal data. Following, user preferences are updated, or new ones are added, by dynamically modeling the feedback obtained from user-system interactions. For activated stereotypes, each “key” – “value(s)”-“rating” triplet has also associated a justification (e.g. from which stereotype was obtained) and possible additional parameters (e.g. “context(s)” in which the preference is valid). The triplet “key”-“value(s)”-“parameter(s)” (e.g., “weight”, “confidence”, “context(s)”) is the base for representation of settings in the profile tree.

The profile tree will be much extended as compared to the keys found in the stereotypes, by adding data that cannot be used to represent a group of users in any context and any system, such as *PersonalDetails*, *BiologicalState*, *Relationships*, *ApplicationPrefs*, *DevicePrefs*, *ProductsPrefs*, *ActorsPrefs*, *ProducerPrefs*, *PerformersPrefs*, etc.

5.4 History Data format for Dynamic Modeling

Since dynamic modeling based on short-term data does not make much sense (most of machine-learning methods need sufficiently large amount of data in order to provide meaningful recommendations), dynamic modeling requires application log files in a following form:

"context" - "user action" – positive/ negative label. The format of the log files is described in the section Components Interface for Dynamic Modelling, and examples of all files are provided with the software. User actions for multimedia modeling are in a very simple format:

item name: "the name"

The same format suits e.g. for the retrieval of receipts because they have names.

What do we mean by "sufficiently large amount of data": first, the more the better. Second, experiments with real life TV data have shown that not less than 6-8 positive examples of retrieval of an item is required in order to teach the system when the user is interested in this item.

5.5 Dynamic Modeling tests with real life TV viewing data

5.5.1 Data and pre-processing

The TV viewing data was collected in Finland by Finnpanel OY (Finnpanel: People meter data, <http://www.finnpanel.fi/english.html>) in real life settings, in the normal course of families' lives. Viewing histories were collected during two months. The goal of experiments was to predict future choices of family (that is, of single user and multi-user subsets of family members) based on previous viewing history.

For this study we used data of 20 families with only one TV set in a household; each family consisted of two and more family members (62 test subjects altogether). The raw data contained:

- person IDs and timestamps when this person started/ stopped watching TV (logged via designated control buttons)
- channel numbers and timestamps of switching to these channels, also logged via control buttons

The data pre-processing and the whole learning procedure was done as shown in Figure 3.2. The goal of data pre-processing was to create a list of consecutive viewing sessions in a form: timeslot - time context - social context - list of viewed programs along with their genres, channels and watched percents. Term "social context" denotes a set of people (family members and guests) who watched TV. "Timeslot" denotes time period when TV was switched on and no changes in social context occur: we attempted to generate a new recommendation list each time when social context changed, even though in real life people often join other viewers without checking what's going on in other channels. "Time context" includes both day of week and time of day information. "List of viewed programs" included only programs viewed for more than 40% of there time.

Altogether the data contained about 4000 viewing sessions with almost 9000 TV programs, viewed during these sessions for longer than 40% of the program duration (during one session test subjects viewed 2.2 programs on average). This is a large and unique dataset compare to other works on dynamic modeling. (To the best of our knowledge, very few studies on dynamic modeling of context-dependency of user preferences, including learning of preferences in multi-user settings existed so far).

5.5.2 Experimental protocol and results

Two months of viewing data is a fairly short time period for learning family preferences, thus, we first trained classifiers on the data of first two weeks of each family data, and used the remaining data for incremental learning and testing with this family data as shown in Figure 3.2. We did not use any collaborative filtering; instead, predictions for each family were generated based only on the previous history of this particular family, thus protecting family privacy.

We used purely implicit approach to learning user preferences, and we used CBR and SVM classifiers. We did not use Static Models because they were unknown to us. Testing on the test data was done as follows: from each viewing statement in the test data we took starting timestamp and social context and fed these contexts to the machine-learning methods. Each machine-learning method created a ranked list of recommended programs for the next two hours (we feel it to be long enough for sitting in front of TV). The lists of recommended programs were compared with the list of the actually viewed programs, and for each list accuracy of predictions was calculated. Recommendations were created (and compared with the really viewed programs) for each viewing session of the family test data, and the accuracy of predictions was averaged over the whole test data.

Here we present the accuracy of predictions in a form of “recall at N best”. Recall is one of the typical performance measures of recommender systems. If we denote the total number of available relevant (actually viewed) programs as N_{rel} and the number of relevant programs, included into recommendations as N_{rel_rec} , then recall is calculated according to the formula:

$$R = \frac{N_{rel_rec}}{N_{rel}}$$

“Recall at N best” is recall in the list which includes only N top recommendations.

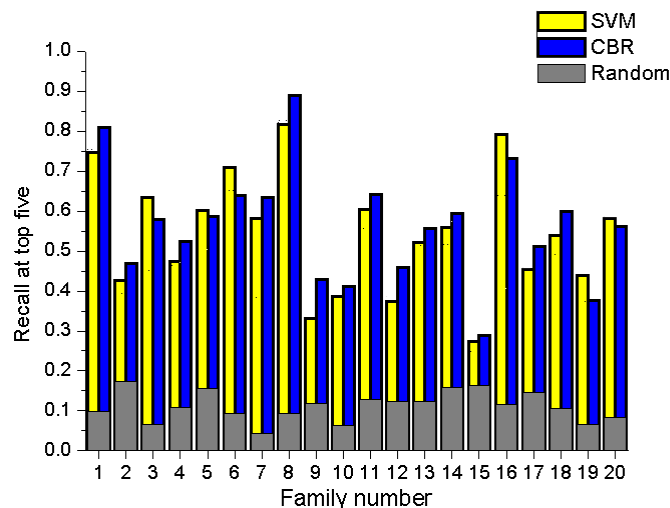


Figure 5.1 Recall at N best by SVM and CBR, and recall of random predictions for comparison

The worst recall was in six families with the shortest viewing histories and in two families with sufficiently long histories (family number 2 and family number 19).

5.5.3 Conclusions

Although the prediction accuracy in our initial tests was not miraculously good, we think that it was not bad either because the data presented the following challenges:

- short period of training data (2 months)
- broad metadata descriptions (e.g. “factual” or “foreign movies”): it is not easy to find detailed metadata on many ongoing TV programs.
- significant change in viewing habits, caused by weather changes (average number of viewing sessions per family during last two weeks of viewing histories was only 1.7 times smaller than number of viewing sessions per family during first six weeks)

Unfortunately it is not easy to compare our results with the state-of-the-art because we are unaware of any other works presenting similar machine-learning experiments with real life data. We can only compare our results with other recommendation systems. For example, collaborative filtering is a famous recommendation method which requires the users to rate the viewed programs as “good”, “bad”, “not so good” and so on, and recommends items rated as “good” by similar users. One state-of-the-art collaborative filtering system reported prediction accuracy among top five recommendations to be in a range 65% - 75% by different methods. Accuracy of our recommendations for the families with sufficiently long viewing histories was 55-80%, but our approach does not require users to rate the programs.

Thus we conclude that our approach to user modeling performed fairly well. Furthermore, in our experiments average recommendation accuracy for multi-user environments was close to that of single-user viewing sessions, despite significantly smaller amount of multi-user training data, which confirms the advantages of learning a joint model of multiple users compare to attempts to build this model by merging static profiles (the works which proposed methods to do it also concluded that merging static profiles does not provide satisfactory recommendations when preferences of users differ significantly).

5.6 FAQ