SmartSlog Manual:
Multilingual Ontology Library Generator for Smart-M3 Platform

March 26, 2012

Contents

List of Abbreviations and Basic Terms

KP: **K**nowledge **P**rocessor.

M3: **M**ulit-device, **M**ulti-vendor, **M**ulti-domain.

SIB: **S**emantic **I**nformation **B**roker.

SmartSlog: **Smart S**pace Onto**log**y Library Generator

SSAP: **S**mart **S**pace **A**ccess **P**rotocol.

Chapter 1

Introduction

SmartSlog [1, 2] is a tool for knowledge processor (KP) development and is a part of Smart-M3 ADK [3–5]. Given an ontology description (in OWL) SmartSlog provides *an ontology library* that allows write KP code in OWL terms of classes, properties, and individuals. It simulates ontological description in local data structures of a given programming language as well as implements all KP-to-SIB and SIB-to-KP operations in these data structures. The usage scheme is shown in Fig. 1.1.

The approach is close to object-RDF mapping libraries of the semantic web, but SmartSlog (i) does not limit itself with object-oriented programming languages and (ii) is primarily oriented to statically-typed compiled languages, where object-RDF mapping is more difficult for implementation than in interpreted languages. SmartSlog library acts as high-level KP interface (KPI), which is in contrast to low-level KPI with RDF triples as basic units in KP-to-SIB and SIB-to-KP operations.

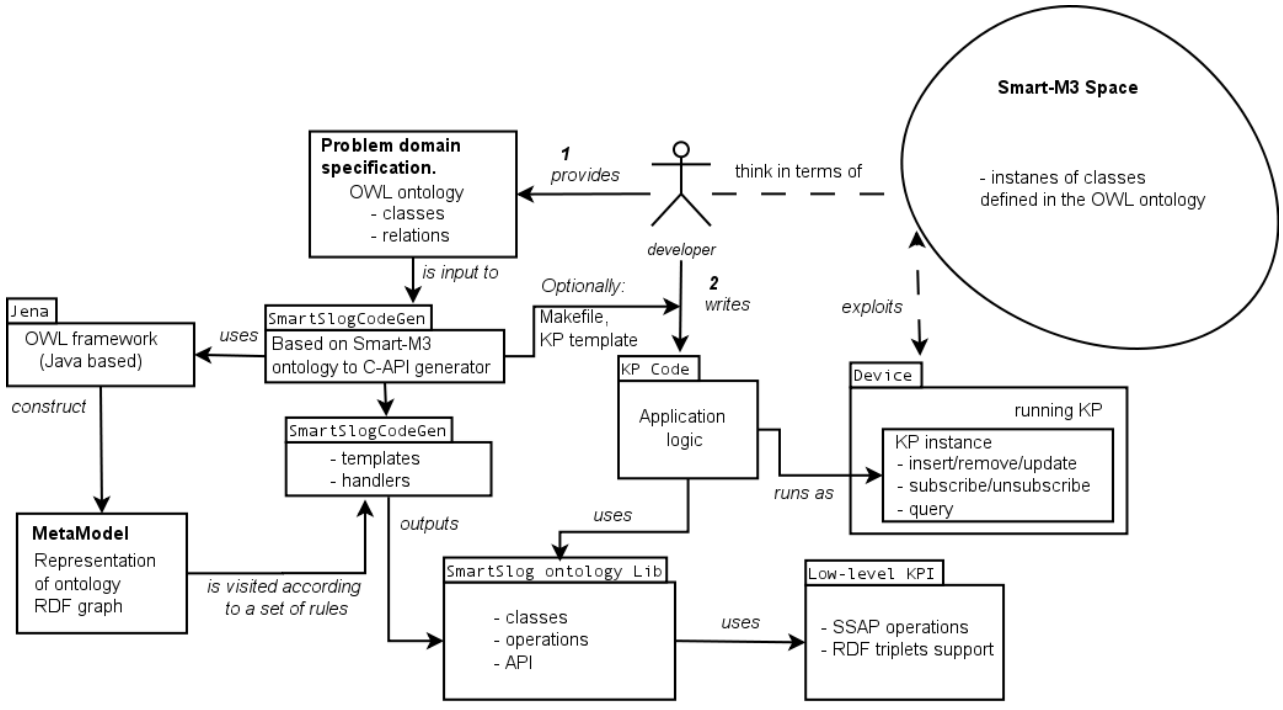

Figure 1.1: SmartSlog usage scheme.

Ontology *library API* is generic, i.e., ontology entities appear as arguments in API functions and the API function names are not tied to any specific ontology. Two API models are implemented: procedural (C is a reference case) and object-oriented (C# is a reference case).

Any ontology entity takes a constant local space, so the memory usage is predictable. Unused ontology entities can be eliminated from the resultant code, so KP deals with a part of ontology. Runtime RDF-OWL mapping uses local triple repository. Complex queries to the SIB may require constructing locally a set of RDF subgraphs. Whenever a triple has been used it is deleted from the triple repository. The latter discipline is adequate for low-capacity devices. When an individual moves from SIB-to-KP or vice versa only affected properties are transferred through the network.

SmartSlog Code Generator (CodeGen) is implemented in Java and calls Jena-based back-end [6,7] for analyzing the input ontologies specified in OWL (the web ontology language). Support of new output language requires appropriate static code templates provided in advance. In addition to the ontology library, a KP code skeleton can be generated optionally, so the programmer can easier start writing the code.

ANSI C ontology library (ANSI C KPlib) is for the wide class of Linux-like platforms. The code is optimized for low-capacity devices. The dependencies are minimal. POSIX thread support is needed for asynchronous subscription, and can be switched off if the target device does not allow threading. Debug mode can be switched on to output auxiliary information in runtime.

C# ontology library (C# KPlib) is for Windows-family OSes with .NET framework. The code is suitable for smartphones and tablets with Windows Phone 7 and Windows 7. Standard Windows-based PC is supported as well (Windows XP, Vista, Windows 7). Thread support is always on.

Subscription is a persistent query to the smart space. SmartSlog supports synchronous and asynchronous modes. The former blocks the execution and KP logic waits for an update of the subscribed data item in the SIB. Synchronous subscription can be hidden or with callbacks. In the hidden variant the synchronization between KP and SIB is done automatically, the background process does not notify the KP logic, and the latter just uses the most actual data available locally. When callbacks are in use any update explicitly notifies the KP logic about new data.

Knowledge pattern is a unique SmartSlog mechanism for defining an abstract OWL ontology subgraph to match it to the global graph in the SIB. The result of pattern application is all objects that satisfy the relations of this abstract subgraph. Also, a local application is possible to filter objects among all ones locally kept. Recent implementation uses iterative WQL-based queries (WilburQL) to SIB. More efficient implementation requires SPARQL support [8] at the SIB side.

Chapter 2

Installation

2.1 CodeGen: Java environment

SmartSlogCodeGen is written in Java, you can use pre-build jar-package with all dependencies or build the generator using Maven.

For pre-build jar-package you need only Java Runtime Environment version 6 or above of Java SE (Standard Edition). You can get it here: `http://www.oracle.com/technetwork/java/javase/downloads/index.html`. The generator's jar-package is here: `http://sourceforge.net/projects/smartslog/files/SmartSlogCodeGen/`.

If you want to build package by yourself you need to get:

1. Java Developing Kit version 6 or above of Java SE (Standard Edition)
   `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
2. Maven version 2.2
   `http://maven.apache.org/download.html`
3. Last version of SmartSlogCodeGen
   `http://sourceforge.net/projects/smartslog/files/SmartSlogCodeGen`

Install Java and Maven, then unpack SmartSlogCodeGen.

To build a runnable jar package go to the directory where the "pom.xml" file is located (SmartSlog-CodeGen/) and type next commands:

```
mvn clean
mvn package
```

To create a runnable jar package with all dependencies bundled type:

```
mvn assembly:assembly
```

To check corretness of the jar package go to the target directory and type:

```
java -jar SmartSlog-CodeGen.jar
```

If you see help information then setup is successfully completed.

2.2 ANSI C Kplib: Linux environment

This section describes how to install ANSI C version of SmartSlog to Linux environment. We assumes you know how to use Linux, for installations you can use *make* utility.

Third-party components

See following documentations and installation instructions for each component. Some of them have binary packages for easily installation (e.g. *deb*) and some of them you can find in official repositories.

KPI_low and its dependencies:

1. Install scew_v1.1.0 or higher, `http://www.nongnu.org/scew/` (e.g., /usr/local/lib/libscew.a and /usr/local/include/scew.h)
2. Install Expat v.2.0.1 or higher, `http://expat.sourceforge.net/` (e.g., usr/lib64/libexpat.so /usr/local/include/expat.h)
3. Install last version (20122010 or higher) of KPI_low, `http://sourceforge.net/projects/kpilow/`
   (e.g., /usr/local/lib/libkpilow.so /usr/local/include/kpilow/kpi_low.h)

Smart-M3 SIB (it is needed only for implementing SS, if you have no running SIB already):

1. Download last version of smart-m3 (`http://sourceforge.net/projects/smart-m3/`)
2. Unpack smart-m3
3. Install following packages (you need to install *sqlite* and *raptor* additionally):
   libwhiteboard (last version)
   piglet_m3 (last version)

SmartSlog installations

Installation of SmartSlog is fully consistent with auto compilation tools. Download last version of SmartSlog from `http://smartslog.sf.net/` Below structure of downloaded archive:

- NEWS - file with news information.
- README - some useful information about release.
- demos - folder with different examples.

- src - folder with SmartSlog sources.
- AUTHORS - project team and contact information.
- COPYING - GNU GPL v2.0 license.
- INSTALL - installations information.
- ChangeLog - last changes list.

Type consequentially *./configure* , *make* and *make install* to install package.

Demos

In the Demo folder you can find three examples:

1. SimpleHelloWorld – "Hello World" simple example.
2. Drinkers – example with double subscription. Two KPs use subscription to get information about each other.

2.3 C# KPlib: Windows environment

SmartSlog C# KPlib needs the following components:

1. Microsoft Framework 4.0 or above
   `http://www.microsoft.com/download/en/details.aspx?id=17851`
2. SmartSlogKPlib_NET_v0.13alpha:
   `http://sourceforge.net/projects/smartslog/files/SmartSlogKPLibNET`
3. KPI_Low wrapper: KpiLowLibWrapper_v0.2alpha
   `http://sourceforge.net/projects/smartslog/files/KpiLowLibWrapper`
4. Visual Studio 2010 (Express or other)
   `http://www.microsoft.com/visualstudio/en-us/products/2010-editions/express`
5. NUnit framework (optional) for testing
   `http://www.nunit.org/index.php?p=download`

First, install Framework, Visual Studio and NUnit (if needed) using their installation programs. Then, unpack SmartSlogKPlib_NET_v0.13alpha. In the directory with unpacked files you can find

- NEWS - file with news information.
- README - some useful information about release.
- Demo - folder with different examples.
- SmartSlog - folder with SmartSlog project.
- SmartSlog.sln - solution file for SmartSlog and testing projects, you can use it to to start developing the KP.

- SmartSlogTesting - project with unit tests, it uses NUnit framework for testing.

There are three examples in the Demo folder.

1. SimpleHelloWorld - ”Hello World” simple example.
2. HelloWorldSubscribe - ”Hello World” example with subscription.
3. Drinkers - example with double subscription. Two KPs use subscription to get information about each other.

Perform the following steps to start KP code development with SmartSlog.

1. Open solution file (SmartSlog.sln) in Visual studio. If you haven't NUnit framework you can unload SmartSlogTesting project from solution, if you use Visual Studio Express version then you can only remove it .
2. Add new project (for example, simple console application) to the SmartSlog solution, create this project in the directory with SmartSlog.
3. For the new created project add reference to the SmartSlog project and ”using” directive (using PetrSU.SmartSlog) in the code.
4. Go to properties of the new project, open ”Build Events” and set Post-build events:

   ```
   xcopy /C /Y  "$(ProjectDir)/../SmartSlog\dlls\*.dll" "$(TargetDir)"
   ```

   Also you can copy dlls manually to the target directory of the project. All this dlls you can find in the KPI_low_windows_v0.2.
5. Now you can start writing the code of your KP.

Chapter 3

Getting Started

3.1 Hello World application

This part shows you how to write a simple “Hello world” example. It will be consist from two KPs. One publishes (KP-publisher) a property and another (KP-consumer) checks it.

Below you can see an ontology. It will be used in this example. The ontology contains two classes (*Universum, World*) and two properties (*has, isA*). In the example will be used only *Universum* class and *has* property.

```
<rdf:RDF xmlns="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:X="http://X#">

    <owl:Class rdf:about="http://X#World"/>
    <owl:Class rdf:about="http://X#Universum">
        <subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
        <subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="http://X#has"/>
                <owl:maxCardinality rdf:datatype=
                        "http://www.w3.org/2001/XMLSchema#nonNegativeInteger">
                    10
                    </owl:maxCardinality>
            </owl:Restriction>
        </subClassOf>
    </owl:Class>

    <owl:DatatypeProperty rdf:about="http://X#has">
        <domain rdf:resource="http://X#Universum"/>
        <range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    </owl:DatatypeProperty>
```

```
    <owl:ObjectProperty rdf:about="http://X#isA">
        <domain rdf:resource="http://X#Universum"/>
        <range rdf:resource="http://X#World"/>
    </owl:ObjectProperty>
</rdf:RDF>
```

Steps of the example:

1. The KP-publisher joins to the smart space and gets *Universum* individuals or creates a one new.
2. Then it set or update property *has*
3. The publisher leaves the smart space.
4. The KP-Consumer joins to the smart space and tryes to get *Universum* individuals.
5. If it has individuals, then it prints a value of the *has* property.
6. The consumer leaves the smart space.

3.2 Hello World in ANSI C

First of all you need to install last version of SmartSlogKPLib ANSI C and SmartSlog CodeGen.

1. Generating ontology.
   For this step you need the 'Hello' ontology, and SmartSlog code generator (see section 2.1).
   Copy ontology to the folder with generator and use this command:

   ```
   java -jar SmartSlog-CodeGen.jar hello.owl -o .
   ```

   It will produce 4 files:
   kp.c - template of the KP;
   hello.c, *hello.h* - files with mapped ontology, it contains ANSI C structures defining classes and properties;
   Makefile -workable template of the Makefile.

2. Preparing templates for KPs.
   Open file *kp.c* and change include path to actual location of generated *hello.h* file. Let's rename *kp.c* to *publisher_kp.c* and copy it to *consumer_kp.c*. You also need to modify generated Makefile for compilation of two KPs.

3. Implementation of publisher KP.
   There is already skeleton of ANSI C code in *publisher_kp.c*.
   You need to use your information about smart space, such as a space name, IP address and a port.
   Intialization of session with SIB (connect to SIB and join to Smart Space):

```
init_ss_with_parameters("X", "194.85.173.9", 10010);
register_ontology();

if (ss_join(get_ss_info(), "KP_1") == -1) {
    printf("Can't join to SS\n");
    return 0;
}
```

After that, we can try to receive individuals from the smart space and get a property or create a new individual. Let's create individual of class *Universum*:

```
individual_t *universum = new_individual(CLASS_UNIVERSUM);

if (universum == NULL) {
    printf("\nError: %s\n", get_error_text());
    return 0;
}
```

and set any identificator (UUID):

```
if (set_uuid(universum, "some_uuid") != 0) {
    printf("\nError: %s\n", get_error_text());
    return 0;
}
```

Now we can set a new value for the property or update it.

```
if (argc<2 || argv[1] == NULL)
    set_property(universum, PROPERTY_HAS, "Hello world");
else
    set_property(universum, PROPERTY_HAS, argv[1]);
```

Then we also try to insert individual to Smart Space

```
if (ss_insert_individual(universum) < 0)
    printf("Individual can not be inserted\n");
else
    printf("Individual inserted\n");
```

and end session (leave Smart Space)

```
ss_leave(get_ss_info());

printf("\nKP leave SS...\n");

return 0;
```

4. Implementation of consumer KP with subscription.
   There is already the same skeleton of ANSI C code in *consumer_kp.c* with session intialization and destroying.

   You need to use your information about smart space, such as a space name, IP address and a port.
   Intialization of session with SIB (connect to SIB and join to Smart Space):

```
init_ss_with_parameters("X", "194.85.173.9", 10010);
register_ontology();

if (ss_join(get_ss_info(), "KP_1") == -1) {
    printf("Can't join to SS\n");
    return 0;
}
```

   After Intialization we need to create individual of class *Universum* with the same UUID (or you can use Knowledge Patterns to find it in Smart Space).

```
individual_t *universum = new_individual(CLASS_UNIVERSUM);

if (universum == NULL) {
    printf("\nError: %s\n", get_error_text());
    return 0;
}

if (set_uuid(universum, "some_uuid") != 0) {
    printf("\nError: %s\n", get_error_text());
    return 0;
}
```

   Now we can get a value for the property to subscribe for it.

```
property_t *property = (property_t *) get_property_type(CLASS_UNIVERSUM, PROPE

if (property == NULL) {
    return 0;
}
```

We have to create a special container for subscription and property list to it

```
subscription_container_t *conatiner = new_subscription_container();

list_t *properties = list_get_new_list();
list_add_data(property, properties);

add_individual_to_subscription_container(conatiner, universum, properties);
```

and subscribe for property list, print current value and wait while it would not be updated by publisher

```
if (ss_subscribe_container(conatiner, false) != 0) {
    printf("\nCan't subscribe\n");
    free_subscription_container(conatiner);
    clean_repositories();
    ss_leave(get_ss_info());
    return 0;
}

const prop_val_t *p_val = get_property(universum, PROPERTY_HAS->name);
if (p_val != NULL) {
    printf("\nNow string is: %s\n", (char *) p_val->prop_value);
}

wait_subscribe(conatiner);
```

Then we print updated value

```
p_val = get_property(universum, PROPERTY_HAS->name);
if (p_val != NULL) {
    printf("\nPublished string: %s\n", (char *) p_val->prop_value);
}
```

and end session (leave Smart Space)

```
free_subscription_container(conatiner);

ss_leave(get_ss_info());
```

3.3 Hello World in C#

1. Generating ontology.
   For this step you need the 'Hello' ontology, and SmartSlog code generator.
   Copy ontology to the folder with generator and use this command:

   ```
   java -jar SmartSlog-CodeGen.jar hello.owl -o . -h CSstatic
   ```

   After the generating you will get 2 files:
   kp.cs - template of a KP;
   hello.cs - the file with mapped ontology, it contains OntologyStructure class.

2. Creating Visual Studio projects.
   Open the solution file of the SmartSlog (SmartSlog.sln), you can find it in SmartSlog release folder. You can see two projects in the solution, if you have not NUnit framework remove or unload SmartSlogTesting project.
   Now add new two projects(ConsoleApplication) to the solution, create them in the directory with SmartSlog and SmartSlogTesting and give them names "PublisherKP" and "ConsumerKP".

3. Preparing projects for developing.
   Remove from new projects files "Program.cs" and add copy "kp.cs" two both project. Also copy file "hello.cs" to the folder with projects and add link to this file from "PublisherKP" and "ConsumerKP". Add a reference to the SmartSlog project. Go to properties of new project, open "Build Events" and set "Post-build" events:

   ```
   xcopy /C /Y  "$(ProjectDir)/../SmartSlog/dlls/*.dll" "$(TargetDir)"
   ```

   Also you can copy dlls manually to the target directory of projects.

4. Working with publisher kp.
   Open 'kp.cs' in the PublisherKP project.
   Add "using" directives:

   ```
   using System.Collections.Generic;
   using System.Linq;
   using PetrSU.SmartSlog;
   ```

   Change the node description in the line:

   ```
   Node node = new Node("Node name");
   ```

   with

   ```
   Node node = new Node("PublisherHelloKP", "X", "127.0.0.1", 10010);
   ```

You need to use your information about smart space, such as a space name, IP address and a port.
Now we can add a code for joining to the smart space:

```
try
{
    node.Join();
}
catch (Exception e)
{
    Console.WriteLine("Can't connect to the smart space. "
            + e.StackTrace);
    Console.ReadLine();
    return;
}
```

After that, we can try to receive individuals from the smart space and get a property or create a new individual.

```
// Get all individuals by class.
List<Individual> individuals = node.GetIndividuals(os.Universum);

// Get first or null if individuals do not exists in the smart space.
Individual universum = individuals.FirstOrDefault();

    string oldValue = null;

     // Check existnce of the individual
     // and create it if it is needed or get property.
    if (universum == null)
    {
        universum = node.CreateIndividual(os.Universum);
        node.Insert(universum);
    }
    else
    {
        oldValue = (string)node.GetProperty(universum, os.Has);
    }
```

Now we can set a new value for the property or update it and leave the smart space.

```
Console.Write("\nSay something: ");

string value = Console.ReadLine();

// If the property is not set - set it,
// otherwise update it using old and new values.
if (oldValue == null)
```

```
        {
            node.SetProperty(universum, os.Has, value);
        }
        else
        {
            node.UpdateProperty(universum, os.Has, oldValue, value);
        }

        node.Leave();
```

5. Working with consumer kp.
   Open 'kp.cs' in the ConsumerKP project. Most changes are equivalent with first KP(PublisherKP). And add "using" directives:

```
using System.Collections.Generic;
using System.Linq;
using PetrSU.SmartSlog;
```

Then we also try to receive individuals from the smart space and write a property value if it is possible.

```
// Create node class, set appropriate data.
Node node = new Node("ConsumerHelloKP", "X", "127.0.0.1", 10010);

// Join to the smart space.
try
{
    node.Join();
}
catch (Exception e)
{
    Console.WriteLine("Can't connect to the smart space.\n"
            + e.StackTrace);
    Console.ReadLine();
    return;
}

// Get all individuals by class from the smart space and
// write their properties value.
List<Individual> individuals = node.GetIndividuals(os.Universum);
foreach (Individual individual in individuals)
{
    string value = (string)node.GetProperty(individual, os.Has);
    Console.WriteLine("Individual universum say: '{0}'", value);
    Console.ReadLine();
}
```

```
        // And levae the smart space in the end.
        node.Leave();
```

6. Testing.
   Start the PublisherKP project, if it connects to the smart space, then you can set or upgrade a property. After that start the ConsumerKP project and it writes value of the property.

Chapter 4

Control of KP library code

This chapter describes available mechanisms that allow controlling the code. SmartSlog does not optimize its mediator library (KPI_low). Instead, SmartSlog optimizes local data structures, the (de)composition (to)from triples, and the way how the mediator library is used. Some of these optimizations are also usable for computers with no hard performance restrictions.

4.1 Preprocessor control directives for ANSI C

Length and timeout settings

The size of internal structures and socket setting you can control with #{define, ifdef} C compiler preprocessor directives. Th following parameters available
for KPI_Low (*kpi_low.h*):

1. **SS_SUBJECT_MAX_LEN** – max length of *subject* filed (char array) in triple structure.
2. **SS_PREDICATE_MAX_LEN** – max length of *predicate* filed (char array) in triple structure.
3. **SS_OBJECT_MAX_LEN** – max length of *object* filed (char array) in triple structure.
4. **SS_RDF_TYPE_MAX_LEN** – max length of *rdf type* filed (char array).
5. **SS_URI_MAX_LEN** – max length of *uri* filed (char array).
6. **SS_SUB_ID_MAX_LEN** – max length of *sub ID* filed (char array).
7. **SS_NODE_ID_MAX_LEN** – max length of *node ID* filed (char array).
8. **SS_SPACE_ID_MAX_LEN** – max length of *space ID* filed (char array).
9. **SS_MAX_MESSAGE_SIZE** – max length of message in communication with SIB (char array).
10. **SS_RECV_TIMEOUT_MSECS** – max timeout for communication with SIB (in milliseconds).

for SmartSlog (*src/ss_func.h*):

1. **KPLIB_UUID_MAX_LEN** – max length of *instance UUID* (char array).

Debug control

Debug mode could be turned on with configure script (see next section) or manualy by setting directives in (*src/utils/kp_debug.h*):

1. **KPLIB_DEBUG_LEVEL** – debug level of library.
2. **KPLIB_DEBUG_ON** – turn on debug mode manualy.

SmartSlog provides folowing debug levels:

```
#define KPLIB_DEBUG_LEVEL 10          /**< Debug level of library */
#define KPLIB_DEBUG_LEVEL_HIGH 1      /**< Highest debug level */
#define KPLIB_DEBUG_LEVEL_AMED 3      /**< Above medium debug level */
#define KPLIB_DEBUG_LEVEL_BMED 7      /**< Below medium debug level */
#define KPLIB_DEBUG_LEVEL_MED 5       /**< Medium debug level */
#define KPLIB_DEBUG_LEVEL_LOW 10      /**< Lowes debug level */
```

You can also find them in *src/utils/kp_debug.h* .

4.2 SmartSlog runtime control for ANSI C

Each ontology entity is implemented as a structure/class of constant size. For ontology with N entities the SmartSlog ontology-dependent part is of size $O(N)$.

In many problem domains, however, the whole ontology contains a lot of classes and properties. First, SmartSlog provides parameters (constants) that limits the number of entities, hence the developer can control the code size. Second, if the KP logic needs only a subset of the give ontology, then SmartSlog allows ontology entity selection/deselection.

Limit for the ontology size

SmartSlog provide constants that limits the number of entities, hence the developer can control the code size. So the developer can select what ontology entities she needs in KP code (or to deselect unneeded). Currently, it is implemented with a simple mechanism based on #{define, ifdef} C compiler preprocessor directives.

Look at the generated file *hello.h* from HelloWorld example:

```
#define INCLUDE_ALL_ONT_ENTITIES 1
#ifdef INCLUDE_ALL_ONT_ENTITIES

#define INCLUDE_PROPERTY_ISA 1
#define INCLUDE_PROPERTY_HAS 1
#define INCLUDE_CLASS_UNIVERSUM 1
#define INCLUDE_CLASS_WORLD 1
#define INCLUDE_CLASS_NOTHING 1
#define INCLUDE_CLASS_THING 1
```

```
#else

#define INCLUDE_PROPERTY_ISA 0
#define INCLUDE_PROPERTY_HAS 0
#define INCLUDE_CLASS_UNIVERSUM 0
#define INCLUDE_CLASS_WORLD 0
#define INCLUDE_CLASS_NOTHING 0
#define INCLUDE_CLASS_THING 0

#endif
```

There is ability to include of exclude all properties and entities or each of them separately.

Local data structures for ontology entities

Also, developer can manualy edit every generated structure. Below presented 2 structures from HelloWorld example.

Data structure for class:

```
CLASS_UNIVERSUM = (class_t *) malloc(sizeof(class_t));
CLASS_UNIVERSUM->rtti = RTTI_CLASS;
CLASS_UNIVERSUM->classtype = strdup("http://X#Universum");
CLASS_UNIVERSUM->properties = list_get_new_list();
CLASS_UNIVERSUM->instances = NULL;
CLASS_UNIVERSUM->superclasses = list_get_new_list();
```

Here developer can change *classtype* . It is strongly recommended do not change other properties.

Data structure for property:

```
PROPERTY_ISA = (property_t *) malloc(sizeof(property_t));
PROPERTY_ISA->name = strdup("isA");
PROPERTY_ISA->about = strdup("http://X#isA");
PROPERTY_ISA->domain = strdup("http://X#Universum");
PROPERTY_ISA->maxcardinality = -1;
PROPERTY_ISA->mincardinality = -1;
PROPERTY_ISA->subpropertyof = NULL;
PROPERTY_ISA->rtti = RTTI_PROPERTY;
PROPERTY_ISA->type = OBJECTPROPERTY;
```

You can control type and cardinality.

Here developer can change *cardinality* (-1 – for infinity), *name* , *about* and *type* . It is strongly recommended do not change other properties.

Of course, all of these fielded could be set in owl specification.

External libraries

Here presented available compilation modes (see the following options for the ./configure script). Developer can switch preferred protocol:
–with-kpilow-access-proto – network protocol for SIB: tcpip or nota (default: tcpip).
Even devices without thread support allow synchronous subscription (default: enable)
–enable-threads – enable threads support (POSIX threads required).
–disable-threads – disable threads support.

The latter case is implemented with a thread that controls updates from smart space and assigns them to the containers. KP is not blocked, and updates come in parallel.

4.3 C# KPI_Low wrapper control

For SmartSlog C# version a wrapper of KPI_Low is used. The wrapper is available here:
`http://sourceforge.net/projects/smartslog/files/KpiLowLibWrapper`

The wrapper allows to manipulate with KPI_Low library using more comfortable API.

SmartSlog version 0.13alpha includes dynamic load library (dll) of 0.2alpha version of the wrapper.

In previous chapter there is information about KPI_Low parameters and constants, it is possible to change their for C# version, but you need to change their in the KPI_Low and in the wrapper. This is necessary because to work with C code from .NET applications marshaling of the data is used and in this case the different strings sizes or other parameters must be equal otherwise it is possible to get exceptions while your application works with the wrapper.

Parameters and constants are in the class *PetrSU.KpiLowLibWrapper.NativeStructures.Constant* it is the internal class of the wrapper and it includes much more parameters and constants from KPI_Low library than in the *kpi_low.h*. Usually there is no need to change their, but in some cases it is may be useful.

Chapter 5

OWL ontologies and SmartSlogCodeGen

SmartSlog manipulates with entities (classes, properties, individuals) instead low-levels triples. You can build ontology that will includes entities and their properties and this ontology can be converted to structures/classes for SmartSlog.

For translating of an ontology to structures/classes SmartSlog Code Generator is used.

SmartSlog Code Generator works with OWL (Web ontology language) ontologies. The generator finds all classes and properties in an graph and builds structures or classes for SmartSlog.

You can see a simple ontology with some classes, object and data-properties here:

```
<rdf:RDF xmlns="http://www.w3.org/2000/01/rdf-schema#"
             ...
          xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

    <owl:ObjectProperty
                rdf:about="http://www.m3.com/2008/02/m3/canonical#drinks">
        <range rdf:resource="http://www.m3.com/2008/02/m3/canonical#Beverage"/>
        <domain rdf:resource="http://www.m3.com/2008/02/m3/canonical#Human"/>
    </owl:ObjectProperty>

    <owl:DatatypeProperty
                rdf:about="http://www.co-ode.org/ontologies/ont.owl#fname">
        <domain rdf:resource="http://www.m3.com/2008/02/m3/canonical#Human"/>
        <range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    </owl:DatatypeProperty>

    <owl:DatatypeProperty
            rdf:about="http://www.co-ode.org/ontologies/ont.owl#numberofdrinks">
        <domain rdf:resource="http://www.m3.com/2008/02/m3/canonical#Man"/>
        <range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    </owl:DatatypeProperty>

    <owl:Class rdf:about="http://www.m3.com/2008/02/m3/canonical#Beer"/>
    <owl:Class rdf:about="http://www.m3.com/2008/02/m3/canonical#Beverage"/>
    <owl:Class rdf:about="http://www.m3.com/2008/02/m3/canonical#Human"/>
```

```
    <owl:Class rdf:about="http://www.m3.com/2008/02/m3/canonical#Man">
        <subClassOf rdf:resource="http://www.m3.com/2008/02/m3/canonical#Human"/>
    </owl:Class>

    <owl:Class rdf:about="http://www.m3.com/2008/02/m3/canonical#Woman">
        <subClassOf rdf:resource="http://www.m3.com/2008/02/m3/canonical#Human"/>
    </owl:Class>
    ...
</rdf:RDF>
```

You can use generator with the following command:

```
java -jar SmartSlog-CodeGen.jar ontology.owl -o ./
```

This produces the code for ANSI C SmartSlog version.
To get a help information type:

```
java -jar SmartSlog-CodeGen.jar
```

The generator has three types of handlers that can be used for generating. The type of the handler can be set with the parameter '-h'. Values for the parameter can be:

1. 'C': C handler for generating ANSI C code (default handler);
2. 'CS': C# handler for generating C# code with regular class;
3. 'CSstatic': C# handler for generating C# code with static class;

5.1 C# classes

Each ontology entities for C# version of SmartSlog are represented as a classes:

```
// Only public fields and properties are shown
public class OntologyClass
{
    public string ClassType;
    public List<Property> Properties;
    public Property GetPropertyByName(string name);
    public List<OntologyClass> Superclasses;
}

public class Property
{
    public bool IsObject();
    public string Name;
    public List<Property> ParentProperties;
    public int MaxCardinality;
    public int MinCardinality;
}
```

After generating you will get a ⟨*name*⟩`.cs` (by default the name is same as name of file with ontology) file with OntologyStructure class. This class is used as container for ontology entities. You can get access to entities using properties of this class.

Below in the text you will see a part of generated code: a part of regular OntologyStructure class and a list of "#define" directives, you can remove "#define" or undefine some entities to exclude them if they are not necessary for you:

```
#define INCLUDE_PROPERTY_NUMBEROFDRINKS
#define INCLUDE_PROPERTY_FNAME
#define INCLUDE_PROPERTY_DRINKS
...
#define INCLUDE_CLASS_THING
#define INCLUDE_CLASS_WOMAN

    public class OntologyStructure
    {
        public OntologyStructure(Node node)
        {
            if (node == null)
            {
                throw new System.ArgumentNullException("node");
            }
            RegisterOntology(node);
        }

#if INCLUDE_CLASS_HUMAN
        public OntologyClass Human {get; private set;}
#endif
#if INCLUDE_CLASS_BEVERAGE
        public OntologyClass Beverage {get; private set;}
#endif
...
#if INCLUDE_PROPERTY_FNAME
       public Property Fname {get; private set;}
#endif
#if INCLUDE_PROPERTY_DRINKS
       public Property Drinks {get; private set;}
#endif
...
}
```

To use ontology entities in a code you need to add to a project a file with OntologyStructure class, change, if it necessary, the namespace of this class, add "using" directive, create an instance of the class and register entities in a Node:

```
    using GeneratedCode; // Namespace of the OntologyStructure class
    ...
    Node node = new Node(nodeName, smartSpaceName, address, port);
```

```
OntologyStructure os = new OntologyStructure(node);
```

To register entities in the Node we pass instance of the Node class as parameter of OntologyStructures constructor. The Node class is main class to access smart space, to manage classes, to manipulate with remote properties and for some other actions.

After this you can work with entities:

```
// Create an individual
Individual ind = node.CreateIndividual(os.Human);
// Set some property
node.SetProperty(ind, os.Fname, "name");
```

As you can see above you can generate a regular and static class with ontology entities. A difference is in what you do not need to create the OntologyStructure class. This way give you possibility to get global access to ontology entities in your project. By default the class is created as public class, in some cases it is useful to make it as internal for you project.

As for the regular class you need to register entities in the Node class before using them, but for this action you need to use "RegisterOntology" function:

```
Node node = new Node(nodeName, smartSpaceName, address, port);
OntologyStructure.RegisterOntology(node);

// Get all individual of Human class from smart space
List<Individual> individuals = node.GetIndividuals(OntologyStructure.Human);
```

5.2 ANSI C structures

Each ontology entities for ANSI C version of SmartSlog are represented as structures:

```
typedef struct class_s {
        int rtti;               /**< Run-time type information. */
        char *classtype;        /**< Type of class, name. */
        list_t *superclasses;   /**< List of superclasses. */
        list_t *oneof;          /**< Class oneof value (OWL oneof). */
        list_t *properties;     /**< Properties list for class. */
        list_t *instances;      /**< List of individuals. */
} class_t;

typedef struct property_s {
        int rtti;               /**< Run-time type information. */
        int type;               /**< Property type: object, data. */
        char *name;             /**< Name of property. */
        char *domain;           /**< Property domain. */
        char *about;            /**< About field. */
        list_t *subpropertyof;  /**< Parent properties list. */
        list_t *oneof;          /**< Propertie's oneof value(OWL oneof). */
        int mincardinality;     /**< Minimal cardinality. */
```

```
        int maxcardinality;      /**< Maximum cardinality. */
} property_t;
```

After generating you will get two files ⟨*name*⟩`.c` and ⟨*name*⟩`.h` (by default the name is same as name of file with ontology) with all entities from ontology and implemented function "register_ontology", this function is needed to be called before using entities, it creates all entities structures and register them in the inner repository.

In the ⟨*name*⟩`.h` file there are "#define" directives, you can manipulate with them to exclude or include entities:

```
#define INCLUDE_PROPERTY_NUMBEROFDRINKS 1
#define INCLUDE_PROPERTY_FNAME 1
#define INCLUDE_PROPERTY_DRINKS 1
...
#define INCLUDE_CLASS_HUMAN 1
#define INCLUDE_CLASS_BEVERAGE 1
```

To use ontology entities in the code you need to add include ⟨*name*⟩`.h` file and call "register_ontology" function, then you can use entities:

```
#include "<name>.h"
...
    init_ss_with_parameters("X", "127.0.0.1", 10010);
    register_ontology();
    ...
    // Create an individual
    individual_t *ind = new_individual(CLASS_HUMAN);
```

5.3 Multiple ontologies

You can generate file that includes many ontologies:

```
java -jar SmartSlogCodeGen
    ontology1.owl ontology2.owl ontology3.owl -o .
```

In the generated file will be all entities from all given ontologies. If ontologies have equals entities only one will be included to the result file.

5.4 The filtering of entities

When you use the generator you can set a filter file(s):

```
java -jar SmartSlogCodeGen
    ontology1.owl ontology2.owl  -f filter1.txt -f filter2.txt -o .
```

The filter file contains URIs of entities that will be used for the generating. Such file has very simple format:

```
--- Classes ---
http://www.xfront.com/owl/ontologies/camera/#Range
http://www.xfront.com/owl/ontologies/camera/#Viewer
--- Object properties ---
http://www.xfront.com/owl/ontologies/camera/#viewFinder
http://www.xfront.com/owl/ontologies/camera/#part
--- Data properties ---
http://www.xfront.com/owl/ontologies/camera/#f-stop
--- Object properties ---
http://www.xfront.com/owl/ontologies/camera/#compatibleWith
...
```

You can write it by youself, but it is possible to use a plug-in for the Protege (version 4.1 or above). You can get the Protege here: `http://protege.stanford.edu`
The plug-in is available here: [**XXX: TODO downlaod link**]

If you are working with sources of the plug-in, then read README and INSTALL files. If you download the pre-build jar package, then:

1. Copy it to the Protege plugin directory;
2. Start the Protege;
3. Go to '*Windows*' - '*Tabs*' - and check '*Entities chooser*';

Using this tab you can check/uncheck entities and save/load them to/from file.

5.5 Pure OWL standard and Jena's features

SmartSlog code generator now supports only pure OWL ontologies that serialized in RDF/XML format. But in some cases Jena can correct understand ontology structures that doesn't use OWL format:

```
<rdf:Property rdf:ID="testProp">
  <range xmlns="http://www.w3.org/2000/01/rdf-schema#"
         rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</rdf:Property>
```

Here where is no OWL construction of the data property, but Jena can parse it correctly using "range". It also works for object properties, but if you remove "range", it will not be able to determine the type and generator skips property. It is better working with OWL constructions: 'DatatypeProperty' and 'ObjectProperty'.

Sometimes after converting from an other format you can get construction such as:

```
<Class rdf:about="http://www.m3.com/2008/02/m3/canonical#Human">
   <can:fname rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
   <can:lname rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</Class>
```

In this case Jena can't correctly determine property type and always set it as object. You need to remove or modify such structures by hand.

Another thing that can mislead you is intersection and disjunction of property's domain and ranges, for example:

```
<owl:DatatypeProperty rdf:ID="title">
  <rdfs:domain rdf:resource="#Post"/>
  <rdfs:domain rdf:resource="#Comment"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string/>
</owl:DatatypeProperty>
```

It is possible to decide that this property can be set for Post and Comment, but it is incorrect, because by default this construction should be interpreted as a intersection. Instead you should use owl:unionOf as analogous to logical disjunction. The OWL specifications gives following example about multiple domains (`http://www.w3.org/TR/owl-ref/#domain-def`):

```
<owl:ObjectProperty rdf:ID="hasBankAccount">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Person"/>
        <owl:Class rdf:about="#Corporation"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
</owl:ObjectProperty>
```

When you work with Protege (`http://protege.stanford.edu/`) it points out that ranges and domains for property will be use as intersection. You can see the word "intersaction" in brackets near "Ranges" and "Domains", so set data for you properties correctly.

Chapter 6

Complexity of operations

Transactions are supported on the SIB side. In SmartSlog terms it means that all triples, sent in one message to SIB would be processed as one transaction. The limit of the amount of the triples depends on limit of the message size only.

So, all API functions of SmartSlog could be devied to atomic (sended one message to SIB) and non-atomic (two ore more messages).

6.1 Atomic and non-atomic operations

Functions without patterns for setting property in SS, checking existance, updating properties, inserting objects and removing all entities are fully atomic. It compose only one request.

Functions fot getting property from SS: one request to find property, second request for properties with object type to find individual if not exists in local repository.

Function for update individual (*ss_update_individual*) works different. The first request deletes individual from Smart Space and the second request inserts new individual.

Searching individuals in Smart Space is iterating procedure. It creates one request for each object property of all individuals, active in query processing. So, for query for individual without object properties only one request would be sent. Else for each object property would be sent one more request. If individual, corresponding some of object property has object properties too also one more request would be sent for them.

6.2 Triples in communication with SIB

Any SS function assumes constructing of the list of the triples and then transformation it to XML message (KPI_Low library). How mutch triples would be sent for each operation?

All functions for working with properties (*ss_set_property*, *ss_get_property*, e.g.) are send only one triple for each request.

All functions for working with objects (individuals or classes) assumes transformation of this entity to list of the triples: one triple for the type of the object and one triple for each property.

Asynchronous subscription sends one triple to get indicators. The timeout of subscription see in chapter.

Chapter 7

Subscription operation

Subscription is used to synchronize a local data with a data in a SIB. Using the SmartSlog you can subscribe to properties of an individual. For subscription you need an individual, a set of properties and a subscription container. You can use one container for different individuals and their properties. For one container creates one subscription and one TCP-socket, it doesn't depends from how many individuals and properties are in the container.

Subscription consists of next steps:

1. First you need to set correspondence between individuals and properties. Such kinds of correspondences are organized using subscription container. Into subscription container you can add pairs: *individual - list of properties*.

2. Then you need to subscribe the container. The container can be subscribed synchronously and asynchronously. For each containers created only one subscription regardless how many properties and individuals are in the container.

3. If the container subscribed successfully, then first synchronization is passed, after this step all subscribed properties for current individuals are blocked for changing.

4. Processing of the subscription depends of the type (synchronous/asynchronous)

   (a) If the type is synchronous, then after subscription the container is expected to be called with a function to check the container. If this function is called with the container as a parameter, then the current thread is blocked and and the function waits data from the SIB. If the data is received, then the function updates the container and finishes work.

   (b) If the type is asynchronous, then the updating of the container is in background thread and you can work with subscribed data in the real time.

5. The last step is an unsubscription of the container.

When you subscribe some individual's properties you can't change it manually, properties are updated only when current data changes in the SIB.

7.1 Synchronous and asynchronous subscription

Subscription can be synchronous and asynchronous. When you use synchronous subscription you need to call special function to start waiting a new data from the SIB, this function blocks current thread and waits the data. When the new data have been received and the container have been updated the function returns control to the current thread. Synchronous subscription doesn't create any other threads, it works in the thread from which it was called.

Asynchronous subscription works in the other thread and doesn't blocks the current thread. All containers that was subscribed as asynchronous are updated in the background process.

7.2 Callbacks

To track a subscribed data changes you can use callbacks (or C# events/delegates, now this feature is implemented only for C# version). You set a needed function as the callback after creating the subscription container. This function will be called after updating the container, to the function will be passed the current subscription container, individuals and updated properties.

Remember that callbacks work synchronously, that's why you need to write simple callbacks function, it is more important for asynchronous subscription because callback stops subscription checking process for other asynchronous containers.

To use callbacks you need to write function with signature as delegate:

```
delegate void UpdateHandler(SubscriptionContainer container,
    SubscriptionEventArgs ea);

//For example:
void CheckSubscribe(SubscriptionContainer container,
    SubscriptionEventArgs ea)
{ ... }
```

And add it as handler to the update event:

```
container.UpdateEvent +=
    new SubscriptionContainer.UpdateHandler(CheckEvent);
```

7.3 ANSI C version of subscription

Subscription steps for ANSI C version of SmartSlog:

1. First of all you need to create a subscription container:

```
subscription_container_t *container = new_subscription_container();
```

2. Then you need to add individuals and properties to the container:

```
list_t *properties = list_get_new_list();
list_add_data(property, properties);

add_individual_to_subscription_container(container, individual, properties);
```

3. And subscribe the container:

```
if (ss_subscribe_container(container, false) != 0) {
    printf("\nCan't subscribe\n");
    ...
 }
```

4. This step is a tracking subscription process, this step depends from subscription type (synchronous or asynchronous), but if you use synchronous subscription you need to check it using function "wait_subscribe(subscription_container_t*):

```
wait_subscribe(container);
```

For ANSI C version you can choice subscription type using a second parameter of the **ss_subscribe_container** function, *true* means subscription is asynchronous, *false* means synchronous:

```
subscription_container_t *container = new_subscription_container();
...
if (ss_subscribe_container(conatiner, true) != 0) { // asynchrnous subscription
    printf("\nCan't subscribe");
    ...
}
```

5. When subscription is no longer needed you can unsubscribe the container:

```
ss\_unsubscribe\_container(conatiner);
```

7.4 C# version of subscription

For C# version's steps are the same as for ANSI C version:

```
// Create container
SubscriptionContainer container = new SubscriptionContainer();

// Add to the container individual with properties
ListList<Property> properties
    = new List<Property> { NameProp, SurnameProp };
container.Add(individual, properties);

// Subscribe the container
if (node.Subscribe(container) == false)
```

```
{
      Console.WriteLine("Can't subscribe");
     ....
}

// Waiting new data (for synchronous subscription)
node.WaitSubscribe(container);

// Unsubscribe container
node.Unsubscribe(container);
```

For C# you can choice a type of the subscription while you create the subscription container, by default synchronous subscription is used:

```
// Container for the synchronous subscription - default
SubscriptionContainer container = new SubscriptionContainer();

// Container for the synchronous subscription - explicit choice
SubscriptionContainer container =
    new SubscriptionContainer(SubscriptionType.Synchronous);

// Container for the asynchronous subscription
SubscriptionContainer container =
    new SubscriptionContainer(SubscriptionType.Asynchronous);
```

Chapter 8

Knowledge Patterns

8.1 Overview

A data model that allows defining ontological objects. It is a base tool for searching and filtering the smart space content. Developer defines objects by composing a pattern for filtering locally available objects or for searching new objects in the smart space.

A pattern for an object contains only a subset of properties needed for filtering/searching. In filtering, these properties are compared with local objects. In searching, these properties are used to find appropriate objects. The result is an object that contains values only for the given properties. This way reduces the amount of data to keep, process, and transfer, even if the actual objects have many properties.

Figure 8.1 represents a particular case of the ontology instance graph. Objects A, B, C and D with datatype and object properties are related to each other using object properties. Instances of these objects are stored in the smart space and locally at the KP.

Our pattern-based approach admits both local (KP) and SIB implementation. Due to the recent restrictions of Smart-M3 SIB SmartSlog implements patterns only on the local side.

Filtering is used for transferring/delivering necessary parts of objects to/from the smart space. As a result, KP works locally with a subset of properties required by the KP logic at current time instance. There is no need to load/save all properties from/to the smart space. Searching is used to deliver (search) new objects, existing in SS.

A data model describes how to organize the structure of data and defines how to process them. In this way there are two criteria for pattern evaluation of patterns: correctness of defining objects and efficiency of processing.

For searching, patterns allow to define object by ontological class, UUID, and checked properties (properties that object should have). To determine object more intelligently patterns should be extended to support unchecked properties (properties that object should not have) and conditional properties (with relations like $\leq, \geq, \leq, \geq$). It also could be useful in filtering.

Currently Smart-M3 SIB does not support SPARQL. Therefore, searching lead to transferring a

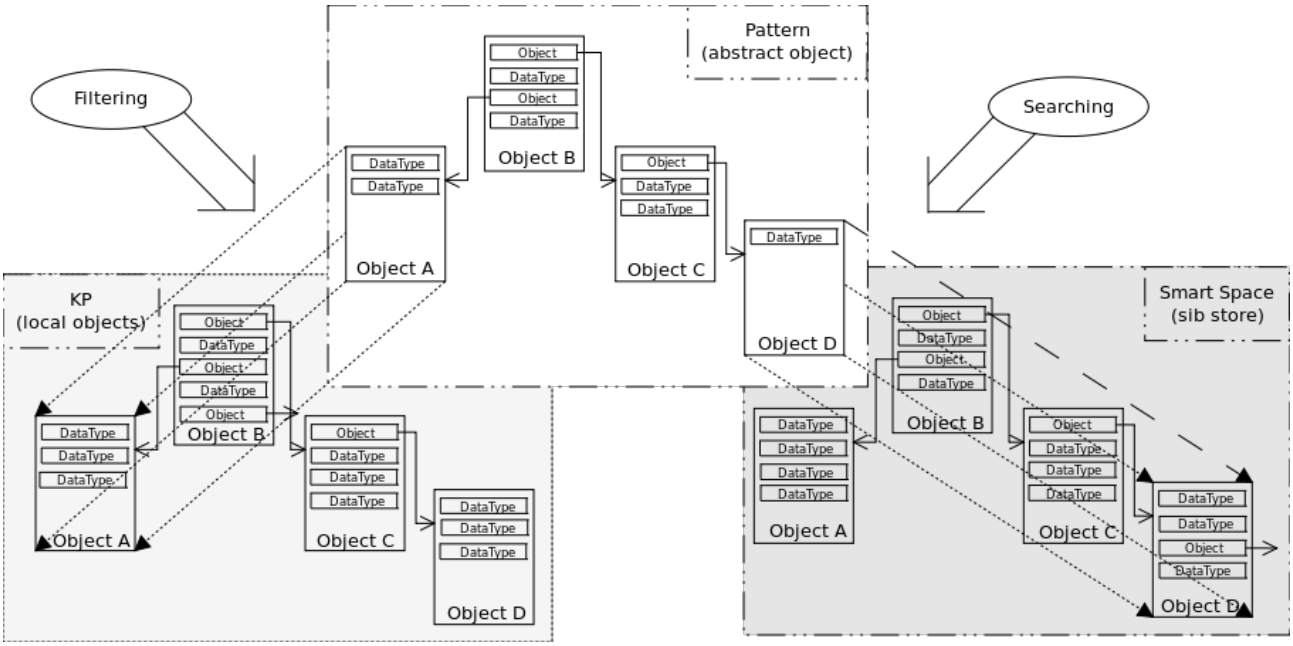

Figure 8.1: SmartSlog patterns for filtration and search.

lot of triples and then to their iterative processing. In both cases, composing a pattern for searching is constructing logical query, which could be optimized. Also the access to properties could be optimized using hash tables. It leads to fast access to necessary properties.

8.2 Usage

Let's consider an social example related to Drinkers demo. We need to find *man* with first name *"Timo"* and second name *"Erricsson"* and has *wife* (object property) with first name *"Alice"* .

First, you need to create a pattern and set all known properties, object property could be individual or pattern too:

```
pattern_t *p_wife = new_pattern(CLASS_MAN);

set_property(p_wife, PROPERTY_FNAME, "Alice");


pattern_t *p_timo = new_pattern(CLASS_MAN);

set_property(p_timo, PROPERTY_FNAME, "Timo");
set_property(p_timo, PROPERTY_LNAME, "Ericsson");
set_property(p_timo, PROPERTY_WIFE,  p_wife);
```

Then we call a search procedure. It step by step compose requests and filter unnecessary individuals. Return value would be a list of individuals, mutched query.

```
    list_t *timos = ss_get_individuals_by_pattern(p_timo);
```

```
        if(*timos == NULL)
        {
                printf("There are no such individuals");
                return NO_INDIVIDUALS;
        }
```

We can iterate throw list, check count of returned individuals and clarify query if needed. Or we can take the first individual and work with him:

```
        individual_t *timo = new_individual(CLASS_MAN);

        list_head_t *pos = NULL;
        list_for_each(pos, &timos->links) {
                list_t *node = list_entry(pos, list_t, links);
                timo = (individual_t *)node->data;
                break;
        }
```

Chapter 9

Cross-platform Code

9.1 KP in ANSI C for Linux environment

SmartSlog is primarily oriented to low-performance (embedded) devices [9] and uses a limited subset of ANSI C [10]. The only dependence is KPI_Low library that also use pure ANSI C code. So ANSI C version could be used on any device with c compiler.

9.2 KP in ANSI C for Windows environment

There is no ready version of ANSI C SmartSlog for Windows OS, for example as a Visual Studio project, but it possible to port SmartSlog as dynamic load library (dll) for Windows. Also you can build and use SmartSlog under Linux emulators/environments for Windows such as MinGW or Cygwin.
Use this links to get more information:
`http://www.cygwin.com`
`http://www.mingw.org`

9.3 KP in ANSI C for Qt/C++

You can use SmartSlog ANSI C version to develop C++/Qt application.
All what you need is adding to a .pro file information about location of the library and headers files, for example:

```
INCLUDEPATH += -I/usr/local/include
LIBS += -L/usr/local/lib -lsmartslog
```

And add '*include*' directive in your code:

```
#include <smartslog/generic.h>
```

Also it possible to use QML (Qt Meta-Object Language) to build powerful interfaces for you KPs. You can reed more about this language and how to use it in your applications here:
`http://doc.qt.nokia.com/4.7-snapshot/qdeclarativeintroduction.html`
s

Chapter 10

Advanced examples

10.1 GPS Locations

Smart space, as any other space can be defined by using set points with their IDs in the space. As such identifiers are the geographical coordinates, namely latitude, longitude and altitude.

Let there be a set of KP-publishers, each of which has its geographical coordinates are determined in real time using the GPS-receiver, is also a KP-consumer, which detects the presence of a KP-publisher in the smart space. As soon as the KP-publisher falls in the scope of the smart space, he registered it and will publish its geographic coordinates, and out of scope, it removes its information from the smart space, thus leaving it. KP-consumer performs its function by implementing a subscription to KP-publisher.

Implemented a demo performs a functional part KP-publisher: namely, KP publishes a GPS-point, which determines the geographical position of KP-publisher in the SIB (imitation of input) and removes it after some time (imitation of output), this operation is repeated. Functional of KP-consumer is implemented manually, using a tool ssls, which allows us to analyze the contents of SIB, also perform the subscription to properties, individuals, classes. The demo is implemented in C# using SmartSlog for .NET platforms.

Below is the ontology satisfying the specification of OWL and describing the spatial location objects, using WGS84 as a unified system of coordinates:

```
<rdf:RDF xmlns="http://www.w3.org/2000/01/rdf-schema#"
         ...
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

 <owl:ObjectProperty
       rdf:about="http://www.w3.org/2003/01/geo/wgs84_pos#location">
   <range rdf:resource="http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing"/>
 </owl:ObjectProperty>

 <owl:DatatypeProperty
       rdf:about="http://www.w3.org/2003/01/geo/wgs84_pos#alt">
   <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
   <range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
```

```
    <domain rdf:resource="http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty
        rdf:about="http://www.w3.org/2003/01/geo/wgs84_pos#lat">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
    <range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <domain rdf:resource="http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty
        rdf:about="http://www.w3.org/2003/01/geo/wgs84_pos#lat_long">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
    <range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty
        rdf:about="http://www.w3.org/2003/01/geo/wgs84_pos#long">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
    <range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <domain rdf:resource="http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing"/>
  </owl:DatatypeProperty>

  <owl:Class rdf:about="http://www.w3.org/2003/01/geo/wgs84_pos#Point">
    <subClassOf rdf:resource="http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing"/>
  </owl:Class>
  ...
</rdf:RDF>
```

On this ontology SmartSlog Code Generator generates a file ”GpsOntology.cs” with OntologyStructure class. This class is used as container for ontology entities. You can get access to entities using properties of this class.

Next will be described in the code file ”GpsKPPublisher.cs”, which implements the core logic of KP-publisher. There will also be the results of the utility ssls for some action of the KP and commands subscribe to the class using the same tools.

Initializes a new instance of the Node class and register the current ontology:

```
Node node = new Node(KPName, SmartSpaceName, SibAddress, SibPort);
OntologyStructure.RegisterOntology(node);
```

With the utility ssls can look that appeared in the SIB after commands execution (running inside ”ls” command):

```
ns_6:Point,rdf:type,rdfs:Class
ns_6:Point,rdfs:subClassOf,rdfs:Resource
ns_6:alt,rdf:type,rdf:Property
ns_6:lat,rdf:type,rdf:Property
ns_6:long,rdf:type,rdf:Property
ns_6:lat_long,rdf:type,rdf:Property
```

From the ssls dump can be seen that in a smart space registred class Point and four properties. n_6 this namespace described by a xmlns declarations enclosed in an opening rdf:RDF tag of the current ontology.

Create new individual of class Point, described in the above ontology and populates a point using different values for properties Alt, Lat, Long and LatLong (values for these properties are generated randomly as a function of):

```
point = node.CreateIndividual(OntologyStructure.Point);
PopulateLocally(point);
```

This is followed by a function call SimulateActivity with two parameters node, point. You can see a description of this function here:

```
private static void SimulateActivity(Node node, Individual individual)
{
    while (isCancelled == false)
    {
        ...

        bool result = JoinAndPublish(node, individual);

        Thread.Sleep(rnd.Next(MaxSleepingTime) * 1000);

        if (result == true)
        {
            CleanAndLeave(node, individual);
        }

        Thread.Sleep(rnd.Next(MaxSleepingTime) * 1000);
    }
}
```

After calling this function is connected to the smart space and registers it in the individual's point. Then sleeps for some time, removes the individual of SIB and leave the smart space.

Using ssls can display a list of properties (set earlier in the function PopulateLocally) the individual's "point" after performing "JoinAndPublish" (running inside ls command):

```
3277dcf4-cd55-42d0-bcad-08541a7846f9, rdf:type, ns_6:Point
3277dcf4-cd55-42d0-bcad-08541a7846f9, ns_6:lat_long,"108066222,349417316"
3277dcf4-cd55-42d0-bcad-08541a7846f9, ns_6:alt,"219349190"
3277dcf4-cd55-42d0-bcad-08541a7846f9, ns_6:lat,"108066222"
3277dcf4-cd55-42d0-bcad-08541a7846f9, ns_6:long,"349417316"
```

These changes in the SIB during the execution of function SimulateActivity (registration and removal of the individual) can be traced if you do subscribe to the class. By using ssls this can be done as follows:

```
> sub a -v *,rdf:type,ns_6:Point
Subscription ID 7 : ('*', 'rdf:type', 'ns_6:Point')
```

Then automatically we receive by appropriate messages from the utility ssls, with any changes of individuals belonging to the class Point:

```
Subscription for 7
+ 3277dcf4-cd55-42d0-bcad-08541a7846f9,rdf:type,ns_6:Point
Subscription for 7
- 3277dcf4-cd55-42d0-bcad-08541a7846f9,rdf:type,ns_6:Point
Subscription for 7
+ 3277dcf4-cd55-42d0-bcad-08541a7846f9,rdf:type,ns_6:Point
...
```

Running multiple KP observe the following output ssls to subscribe:

```
Subscription for 7
+ 3277dcf4-cd55-42d0-bcad-08541a7846f9,rdf:type,ns_6:Point
Subscription for 7
+ dd44e27a-7e5f-4d25-9240-24340fb6b4b6,rdf:type,ns_6:Point
Subscription for 7
+ cf6ebee0-f3fe-4ea2-aae6-d323f63e7c0a,rdf:type,ns_6:Point
Subscription for 7
- dd44e27a-7e5f-4d25-9240-24340fb6b4b6,rdf:type,ns_6:Point
Subscription for 7
+ 2739ae75-d361-405b-aa59-f1cc792035ec,rdf:type,ns_6:Point
Subscription for 7
- cf6ebee0-f3fe-4ea2-aae6-d323f63e7c0a,rdf:type,ns_6:Point
...
```

Chapter 11

Low-level operations

SmartSlog is a high-level tool for constructing KPs. It means developer manipulates with OWL-objects (classes, properties and individuals) while writing KP logic. But sometimes it is necessary to use low-level triples simultaneously with OWL-objects. This chapter describes how to use low-level triples in SmartSlog throw KPI_Low.

11.1 Using KPI_Low in ANSI C

Let's consider an example based on SmartConference's RDF-xontology []. There are set of triples from Projector ontology presented in table 11.1. We have to work with this triples to save integrity of RDF-triples in SIB.

Table 11.1: Part of SmartConference's Projector RDF-ontology

Subject	Predicate	Object
'projector'	'show'	'presentation'
'projector'	'IP'	IPaddress
'NOfSlides'	'is'	number
'presentation'	'has'	'URI'
'current_slide'	'is'	number

SmartSlog maps OWL-object to triples. And subject is an *UUID* of individual would transformed to triples. So according example we should create at least 4 classes: projector, NOfSlides, presentation and current_slide. But we want to work with two classes only: Projector

```
<owl:Class rdf:about="http://X#Projector">
    <subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
 </owl:Class>

<owl:DatatypeProperty rdf:about="http://X#IP">
```

```
    <domain rdf:resource="http://X#Projector"/>
    <range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:about="http://X#show">
    <domain rdf:resource="http://X#Projector"/>
    <range rdf:resource="http://X#Presentation"/>
</owl:ObjectProperty>
```

and Presentation

```
<owl:Class rdf:about="http://X#Presentation">
    <subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
</owl:Class>

<owl:DatatypeProperty rdf:about="http://X#NOfSlides">
    <domain rdf:resource="http://X#Presentation"/>
    <range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://X#URI">
    <domain rdf:resource="http://X#Presentation"/>
    <range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://X#current_slide">
    <domain rdf:resource="http://X#Presentation"/>
    <range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
```

The best practice is to use KPI_Low interface for OWL-objects. To start working with KPI_Low in SmartSlog you should include *kpilow.h* file. This header contains definitions of all API functions of KPI_Low.

Thus we should save additional fields from triples in table 11.1 (e.g. predicate 'is' or object 'URI'):

```
#define ADDITIONAL_PROJECTOR "projector"
#define ADDITIONAL_PRESENTATION "presentation"
#define ADDITIONAL_IS "is"
#define ADDITIONAL_HAS "has"
```

Then we can compose triples according our RDF-ontology:

```
/* Create individuals */
individual_t *projector = new_individual(CLASS_PROJECTOR);
individual_t *presentation = new_individual(CLASS_PRESENTATION);
/* and set them properties */
```

```
    ...

    /* Get necessary properties */
    prop_val_t *p_ip  = get_property(projector, PROPERTY_IP->name);
    prop_val_t *p_show = get_property(projector, PROPERTY_SHOW->name);

    prop_val_t *p_noslides  = get_property(presentation, PROPERTY_NOFSLIDES->name);
    prop_val_t *p_uri = get_property(presentation, PROPERTY_URI->name);
    prop_val_t *p_curr  = get_property(presentation, PROPERTY_CURRENTSLIDE->name);

p_val->prop_value

    /* Compose triple list */
    ss_triple_t * triples = NULL;
    ss_add_triple(&triples,
        ADDITIONAL_PROJECTOR,
        p_show->property->name,
        ADDITIONAL_PRESENTATION,
        SS_RDF_TYPE_URI, SS_RDF_TYPE_URI);
    ss_add_triple(&triples,
        ADDITIONAL_PROJECTOR,
        p_ip->property->name,
        p_ip->prop_value,
        SS_RDF_TYPE_URI, SS_RDF_TYPE_LIT);
    ss_add_triple(&triples,
        p_noslides->property->name,
        ADDITIONAL_IS,
        p_noslides->prop_value,
        SS_RDF_TYPE_URI, SS_RDF_TYPE_LIT);
    ss_add_triple(&triples,
        ADDITIONAL_PRESENTATION,
        ADDITIONAL_HAS,
        p_uri->property->name,
        SS_RDF_TYPE_URI, SS_RDF_TYPE_URI);
    ss_add_triple(&triples,
        p_curr_slide->property->name,
        ADDITIONAL_IS,
        p_curr_slide->prop_value,
        SS_RDF_TYPE_URI, SS_RDF_TYPE_LIT);
```

Then you can insert, remove, update, query and make subscription operation with composed list of triples. See KPI_Low documentation for more information.

11.2 C# KPI_Low wrapper usage

With this wrapper you can manipulate with low-level RDF triples instead with classes, properties and individuals. To start working with the wrapper you need to add a reference to the assembly *PetrSU.KpiLowLibWrapper.dll*, if you have the SmartSlog release, then you also have all needed assembles, otherwise you can download wrapper from `http://sourceforge.net/projects/smartslog/files/KpiLowLibWrapper` it also contains KPI_low dlls that needed for the wrapper. After you setting the reference, add "using" directive to yours code:

```
using PetrSU.KpiLowLibWrapper;
```

Now you can work with wrapper. There is a *Triple* class that represents a triple:

```
Triple triple = new Triple("subject", "predicate", "object",
                                      RDFType.URI, RDFType.Literal);
```

To convert some individual's property to triple, it is need to get an UUID of an individual, URI of a property and a value of the property.

```
Individual man;
...
man.setProperty(nickProperty, "bobik");
...
string subject = man.UUID;
string predicate = nickProperty.Name
string object = man.GetDataProperty(nickProperty);

Triple triple = new Triple(subject, predicate, subject,
                                  RDFType.URI, RDFType.Literal);
```

When you need to manipulate with an object property then use the UUID of an individual also as object and change a type of the object:

```
Individual man;
Individual friend;
...
man.setProperty(friendProperty, friend);
...
string subject = man.UUID;
string predicate = friendProperty.Name
string object = man.GetObjectProperty(friendProperty).UUID;

Triple triple = new Triple(subject, predicate, subject,
                                  RDFType.URI, RDFType.URI);
```

For example, you have a triple in the triple-store that stores a temperature:

```
tempThermometer - hasValue - "12"
```

and you won't or can't to wrap it as an individual, but you need to check a value of it. In this case you can query triple and check the value:

```
// Create an info with data to access smart space.
SmartSpaceInfo spaceInfo = new SmartSpaceInfo(spaceId, ip, port);

// Create a wrapper.
KpiLowWrapper kpilow = new KpiLowWrapper();

// Join to the smart space with ID, this ID is used as KP identifier.
kpilow.Join(spaceInfo, nodeId);

// Create two list for triples.
// One with a triple for a query with a mask as object.
// Other for triple that will be received after query.
List<Triple> requestedTriples = new List<Triple>
{
    new Triple("tempThermometer", "hasValue", Triple.AnyURI,
                    RDFType.URI, RDFType.URI)
};

List<Triple> resultTriples = new List<Triple>();

// Query the triple with temperature.
kpilow.Query(spaceInfo, requestedTriples, resultTriples);

// Get the triple from result.
Triple tempThermometer = resultTriples[0];

// Convert object (the temperature) to integer.
int temp = int.Parse(tempThermometer.Object);

// Check and do something.
if (temp > 10) {
    ...
} else { ... }

...

// In the end you need to leave the smart space.
kpilow.Leave(spaceInfo);
```

Using wrapper you can also *insert*, *remove*, *update* triples and make a *subscription* operation.

Bibliography

[1] D. Korzun, A. Lomov, P. Vanag, J. Honkola, and S. Balandin, "Generating modest high-level ontology libraries for Smart-M3," in *Proc. 4th Int'l Conf. Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM 2010)*, Oct. 2010, pp. 103–109.

[2] "SmartSlog: free development software downloads at SourceForge.net," Dec. 2011. [Online]. Available: http://sourceforge.net/projects/smartslog/

[3] D. G. Korzun, S. I. Balandin, V. Luukkala, P. Liuha, and A. V. Gurtov, "Overview of Smart-M3 principles for application development," in *Proc. Int'l Conf. Artificial Intelligence and Systems (AIS 2011)*, vol. 4. Moscow: Physmathlit, Sep. 2011, pp. 64–71.

[4] J. Honkola, H. Laine, R. Brown, and O. Tyrkkö, "Smart-M3 information sharing platform," in *Proc. IEEE Symp. Computers and Communications*, ser. ISCC '10. IEEE Computer Society, Jun. 2010, pp. 1041–1046.

[5] "Smart-M3: Free development software downloads at SourceForge.net," Release 0.9.5beta, Dec. 2011. [Online]. Available: http://sourceforge.net/projects/smart-m3/

[6] B. McBride, "Jena: A semantic web toolkit," *IEEE Internet Computing*, vol. 6, pp. 55–59, November 2002.

[7] "Jena: Java toolkit for developing semantic web applications based on W3C recommendations for RDF and OWL," Dec. 2011. [Online]. Available: http://jena.sourceforge.net/,http://incubator.apache.org/jena/

[8] E. Prud'hommeaux and A. Seaborne, "SPARQL query language for RDF," W3C Recommendation, Jan. 2008. [Online]. Available: http://www.w3.org/TR/rdf-sparql-query/

[9] M. Barr and A. Massa, *Programming Embedded Systems: With C and GNU Development Tools*. O'Reilly Media, Inc., 2006.

[10] "The ANSI C standard (C99)," ISO/IEC, Tech. Rep., 1999.