Generating Modest High-Level Ontology Libraries for Smart-M3

Dmitry G. Korzun, Alexandr A. Lomov, Pavel I. Vanag
Department of Computer Science
Petrozavodsk State University, PetrSU
Petrozavodsk, Russia
Email: {dkorzun, lomov, vanag}@cs.karelia.ru

Sergey I. Balandin, Jukka Honkola
Nokia Research Center
Nokia
Helsinki, Finland
Email: {sergey.balandin, jukka.honkola}@nokia.com

***Abstract*—Web ontology language (OWL) allows structuring smart space content in high-level terms of classes, relations between them, and their properties. In Smart-M3, a semantic information broker (SIB) maintains the smart space in low-level terms of triples, based on resource description framework (RDF). This paper describes SmartSlog, our solution for constructing Smart-M3 knowledge processors (KPs) that consume/produce smart space content according to high-level ontology terms. The solution is based on the code generation approach. Given an OWL ontology description, the SmartSlog generator maps OWL to the ontology library. It provides 1) API to communicate with SIB and 2) data structures to represent in KP code all ontology classes, relations, properties, and individuals. As a result, the developer easier constructs the KP code, thinking in high-level ontology terms instead of low-level RDF triples. SmartSlog is oriented to ubiquitous systems; the library is modest to the device capacity; it is written in ANSI C, supports even small embedded devices with restricted performance, and allows interoperable applications.**

***Keywords*-Smart spaces; Smart-M3; OWL/RDF ontology; code generator; knowledge processor; low-performance devices**

I. Introduction

A smart space is a virtual, service-centric, multi-user, multi-device, dynamic interaction environment that applies a shared view of resources [1], [2]. Information conforms to ontological representation with subject–relation(predicate)–object triples as in semantic web [3]. Triples are represented using Resource Description Framework (RDF). A number of devices may access information via semantic information brokers (SIBs), which also support information reasoning.

A client application consists of one or more knowledge processors (KPs) running on various user's devices (Figure 1). KPs act cooperatively forming a publish/subscribe system [4]. Each KP can be thought as an agent using the smart space as a shared knowledge space. The KPs produce (insert, update, remove) and/or consume (query, subscribe, unsubscribe) information in a smart space. The smart space access protocol (SSAP) implements the SIB ↔ KP communication, using operations with RDF content as parameters.

A KP may provide information for the smart space and use information provided by other KPs. The information content is not restricted in any way—it may be information relating to the physical environment, to the KPs themselves or anything. Thus, multiple KPs from multiple vendors may share ad-hoc information across numerous domains, enabling cross-domain and cross-platform interoperability.

Application examples include context gathering in meetings [5], meeting room smart space [6], smart home [7], gaming, wellness and music mashup [8] and social networks [9].

Smart-M3 [10] (Multidomain, Multidevice, and Multivendor) is an open software platform [11] that implements the smart space concept. Smart-M3 has been developed by a consortium of companies and within research projects: Artemis JU funded Sofia project (Smart Objects for Intelligent Applications) and Finnish nationally funded program DIEM (Device Interoperability Ecosystem).

Real-life scenarios often involve a lot of information, which leads both to largish ontologies and possibly complex instances that the KPs need to handle. Thus, programming KPs on the level of SSAP operations and RDF triples bring unnecessary complexity for the developers, who have to divert effort for managing triples instead of concentrating on the application logic. The OWL representation of knowledge as classes, relations between classes, and properties maps quite well to object-oriented paradigm in practice (but not so

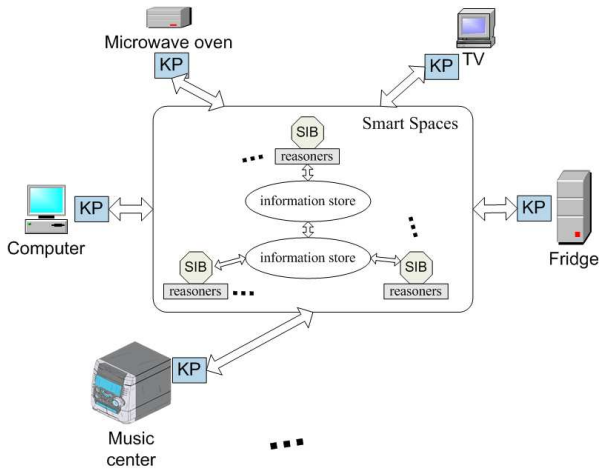

Figure 1. Smart spaces form a publish/subscribe system in a ubiquitous environment: KPs run on various types of computers and devices, the distributed knowledge store supports reasoning over cross-domain information

well in theory). Therefore, it is feasible to map OWL classes into OO classes and instances of OWL classes into objects[1] in programming languages. This approach effectively binds the subgraph describing an instance of an OWL class to an object in a programming language.

This paper describes the SmartSlog ontology library generator tool, our solution for allowing the construction of Smart-M3 KPs by programming with domain concepts that encoded in the relevant ontology.

SmartSlog is an ANSI C library generator for Smart Space ontology [12]. The generator maps an OWL ontology description to ANSI C code (ontology library), abstracting in KP code the ontology and communication with SIBs.

SmartSlog library simplifies constructing KP code. The code manipulates with ontology classes, relations, and individuals using predefined data structures and library API. The number of domain elements in KP code is reduced compared with the low-level triple-based scheme. The API are generic, hence does not depend on concrete ontology; all ontology entities appear as arguments in API functions. Search requests to SIB are written compactly by defining only what you know about the object to find (even if the object has many other properties).

The SmartSlog tool is constructed to take into account the limited resources available on small computers such as mobile and embedded devices. For example, the KP code does not need to maintain the whole ontology as unused entities can be removed. Also, triples are not kept indefinitely as the memory is freed immediately after the use. Furthermore, even if a high-level ontology entity consists of many triples, its synchronization with SIB transfers only a selected subset, saving on communication. These features make it possible to use SmartSlog when developing KPs for small devices—devices that are expected to play a central role in ubiquitous environments.

The rest of the paper is organized as follows. Section II briefly discusses related work Section III introduces the SmartSlog with its architectural and implementation details. Section IV presents generic SmartSlog library API and data structures. Section V describes SmartSlog optimizations. Section VI shows an example of application construction. Section VII concludes the paper.

II. Related Work

SmartSlog is closely related to code generators for C/GLib Smart-M3 KP API and Smart-M3 Python KP API, as they all use a common back-end for analyzing the ontologies and creating a model for code generation (Smart-M3 CodeGen in [11]). Also, the ontology APIs for generated by SmartSlog and the C/GLib generator are very similar. However, the code generated by SmartSlog is more concerned with adequate performance even on low-end devices. For example, dependencies are kept to minimum and memory usage is predictable and bounded.

Ontology based code generation facilities are also provided as part of the Sofia application development kit (ADK) [13] for Java-based KPs. The Sofia ADK is an Eclipse-based toolset for creating smart space applications. The view towards software developer is very similar to the SmartSlog, namely providing programming language view to the concepts defined in an ontology.

Similar ideas also exist in the semantic web world, with projects aiming to provide object-RDF mapping[2] libraries. These libraries are typically not tied to any ontology and implemented in interpreted languages, such as RDFAlchemy [14] in Python or Spira [15] in Ruby. Obviously the approach is very difficult both to implement and to use in statically typed compiled languages such as C, while very convenient in dynamically typed, interpreted languages.

III. Ontology Library Architecture

SmartSlog is built into the base Smart-M3 ontology library generation scheme (Smart-M3 CodeGen in [11]), see Figure 2). For a KP developer, the use scenario consists of two basic steps. First, she (thinking in ontology terms) provides a problem domain specification as an OWL description. The generator inputs the specification and outputs the ontology library. The latter is an interface that eliminates the developer from low-level triple-based details. Second, she uses the library when writing her KP code. The KP logic is implemented in high-level terms of the specified ontology. Note that the developer can easily start coding from KP template and Makefile generated optionally.

An ontology library generator uses a static templates/handlers scheme. Code templates are “pre-code” of data structures that implement ontology classes and their properties. Since names of ontology entities depend on a given ontology, each template contains a tag $\langle \mathrm{name} \rangle$ instead of every proper name. The generator has a set of handlers; each handler transforms one or more templates into final code replacing tags with the names taken from the ontology.

The transformation happens during ontology RDF graph traversal based on Jena OWL framework [16]. The latter constructs a meta-model to represent the graph. The generator comprehensively traverses this model, and those nodes are visited that a handler needs to transform its templates into final code.

Templates and their handlers are device-aware. The dependence is resolved on the level of a mediator library that implements triple-based ontology operations for RDF elements and SIB communication. SmartSlog uses KPI_low [17] as a mediator library, oriented to small embedded devices and ANSI C programming.

[1] These objects only have attributes, but no methods and thus no behavior

[2] In the spirit of object-relational mapping

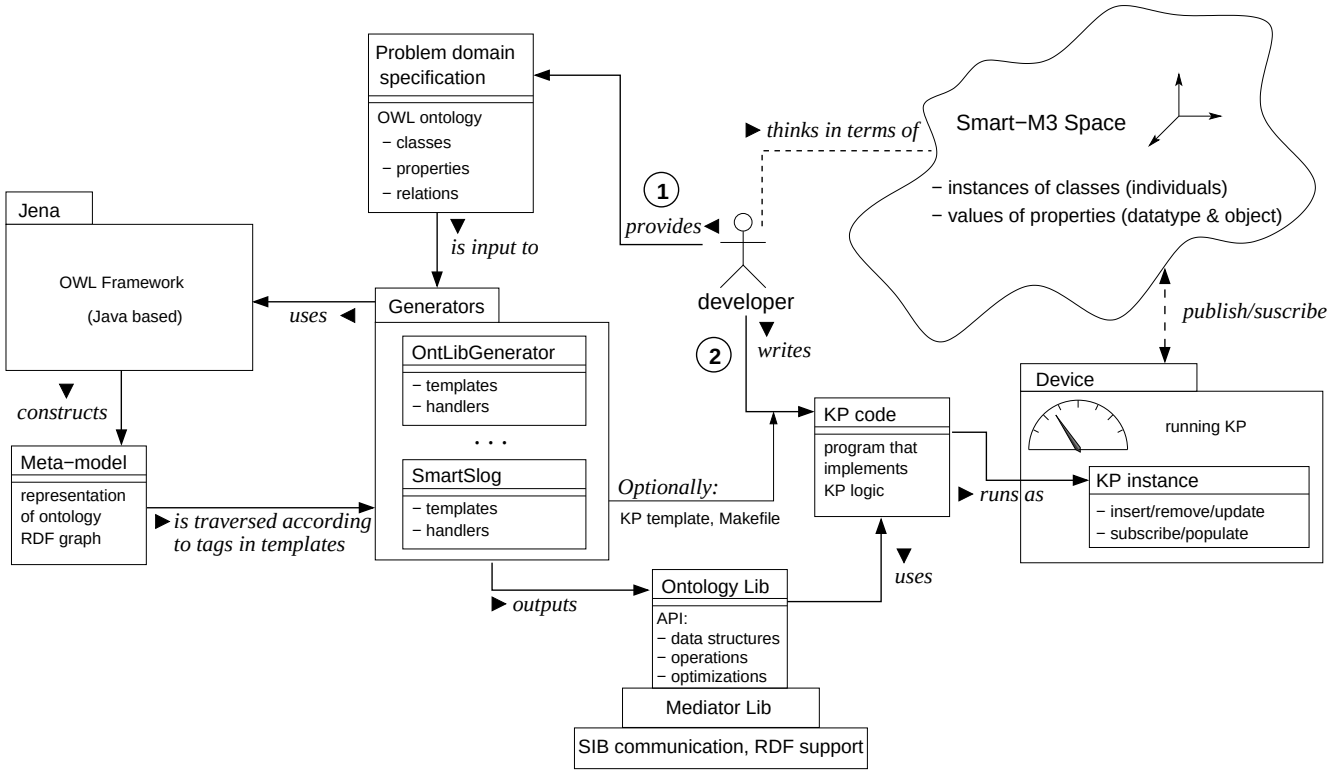

Figure 2. Smart-M3 ontology library generation scheme. Each OntLibGenerator implements own code templates and handlers oriented to a specific mediator library. SmartSlog extends the set of available generators for producing libraries on the top of KPI_low interface (for low-performance devices)

A SmartSlog library consists of two parts: dependent and independent on the given ontology (Figure 3). The SmartSlog generator produces ontology-dependent parts. It is implemented on the top of Smart-M3 CodeGen and uses own ANSI C templates (oriented to KP_low interface). The whole ontology can be represented in several files.

The generator iteratively calls the Jena meta-model. The corresponding templates are loaded and processed, and the final code is generated in files ⟨name⟩.c and ⟨name⟩.h, where ⟨name⟩ is the ontology file name. The code implements all ontology classes and properties as structures in C. Note that the generated code can be optimized further by removing ontology entities unused in the KP.

The ontology-independent part contains API: basic data structures (for generic ontology class, property, and individual) and functions for their manipulation. The code structure is shown in Table I. SmartSlog API uses KPI_low when communicating with SIB. Hence, the ontology-independent part implements all high-level ontology entity transformations to low-level triples and vice versa.

This library division into two parts allows constructing efficient applications. If the ontology changes the ontology-independent part does not require recompiling; it can be shared by several KPs that use different ontology. Ontology-dependent part can be shared by KPs with the same ontology. These cases are typical since multiple smart space applications with different ontology can run on the same device as

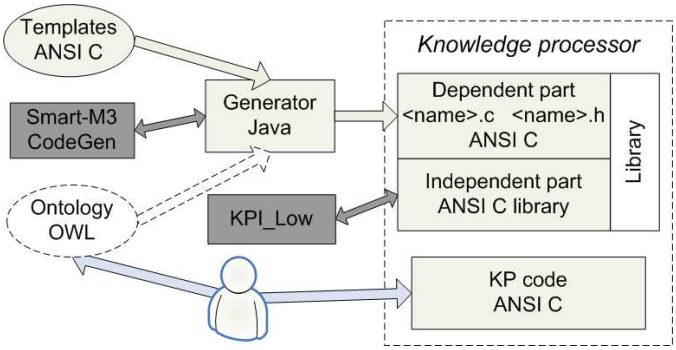

Figure 3. The SmartSlog ontology library architecture: ontology-dependent and ontology-independent parts

well as multiple KPs form one smart space application.

Optionally the generator produces a template for KP code (and Makefile) with function `main()`. In the beginning, it initializes local ontology structures and joins the smart space. In the end, it leaves the smart space gracefully. In between, the developer inserts own code (KP logic).

IV. Library API

SmartSlog API evolves over the generic API of Smart-M3 CodeGen [11]. “Generic” means that API does not depend on ontology: classes, properties, and individuals appear as arguments in API functions. Datatype and object properties

TABLE I
SMARTSLOG CODE STRUCTURE FOR ONTOLOGY-INDEPENDENT PART

Files *.c and *.h	Description	Fnc/Str	LOC/COM
generic.h	Declarations of all API data structures and functions.	0/0	13/21
structures	Base data structures and functions for them.	11 / 4	201 / 214
classes	Manipulation with classes.	20 / 0	318 / 221
properties	Manipulation with properties.	28 / 0	550 / 312
	Sum:	59 / 4	1082 / 768
ss_func	Access to smart space (joining, leaving, ...).	5 / 0	34 / 35
ss_classes	Manipulation with classes in smart space.	15 / 0	344 / 412
ss_properties	Manipulation with properties in smart space.	12 / 0	351 / 383
ss_populate	Population of individuals from smart space.	2 / 0	62 / 107
ss_subscribe	Subscribe containers and functions.	24 / 3	483 / 179
	Sum:	58 / 3	1274 / 1116
kpi_interface	Interface to KPI_low (triple transformation).	9 / 0	401 / 299
utils/*	Auxiliary defs&funcs. Unused directly in KP code.	51 / 3	540 / 641
	Sum:	60 / 3	941 / 940
	TOTAL:	177 / 10	3297 / 2824

Fnc/Str counts the number of functions and structures. LOC/COM was computed by the CCCC tool [18].

are treated similarly. Run-time checking must be performed for arguments.

In SmartSlog, each ontology class, property, and individual is implemented as a C structure (types `property_t`, `class_t`, and `individual_t`). The API has generic functions that handle such data objects regardless of their real ontology content. Currently supported OWL constraints are class, datatypeproperty, objectproperty, domain, range, and cardinality. For example, a class knows all its superclasses, OWL one of classes, properties, and instances (individuals); the implementation is as follows.

```
typedef struct class_s {
  int rtti;       /* run-time type information */
  char *classtype;      /* type of class, name */
  list_t *superclasses; /* all superclasses */
  list_t *oneof;        /* class oneof value */
  list_t *properties;   /* all properties*/
  list_t *instances;    /* all individuals */
} class_t;
```

API functions are divided into two groups: for manipulating with local objects and for communicating with SIB. The first group (local) includes functions for

- classes and individuals: creating data structures and manipulating with them locally;
- properties: operations set/get, update, etc. in local store (also run-time checks for correctness, e.g., cardinality and property values).

For example, creating individual and setting its properties:

```
individual_t *aino = new_individual(CLASS_WOMAN);
set_property(aino, PROPERTY_LNAME, "Peterson");
```

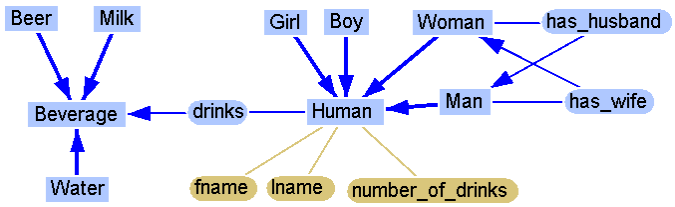

Figure 4. Ontology for humans and their drinks

In this example, the definitions of `CLASS_WOMAN` and `PROPERTY_LNAME` are in the library ontology-dependent part for the ontology shown in Figure 4. (We used GrOwl tool [19]: classes are in blue rectangles, datatype properties are in brown ovals, object properties are in blue ovals.)

The second group (to/from smart space) has prefix "`ss_`" in function names and allows accessing smart space for

- individuals: insertion, removal, and update;
- properties: similarly to the local functions but the data are to/from smart space (it requires transformation to/from triples and calling the mediator library);
- querying for individuals in smart space (existence, yes/no answer);
- populating individuals from smart space by query or by subscription.

For example, inserting an individual and then updating some of its properties:

```
ss_insert_individual(aino);
     . . .
ss_update_property(aino,
    PROPERTY_LNAME, "Ericsson");
```

Subscription needs more discussion. In advance, a subscription container is created to insert those individuals which to subscribe for. Optionally, the container inserts the properties whose values are interested only. Then KP explicitly subscribes for selected properties of selected individuals.

Subscription is synchronous or asynchronous. The former case is simplest; KP is blocked waiting for updates. Even devices without thread support allow synchronous subscription. 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.

Internally, communication with SIB leads to the composition/decomposition of high-level ontology entities from/to triples and calling KPI_low for triple-based data exchange.

SmartSlog API covers all basic primitives of a publish/subscribe system. Compared with Smart-M3 CodeGen that provides similar primitives, SmartSlog API has the following advantages. Smart-M3 CodeGen API depends on glib library, e.g., using list data structures. Low-performance devices do not support glib. In contrast, SmartSlog has no such requirements for underlying libraries. Smart-M3 CodeGen currently does not allow asynchronous subscription important for some smart space applications.

SmartSlog extends generic API by patterns for ontology-based filtering and search. Each pattern is an `individual_t` structure and can be thought as an abstract individual where only a subset of properties is set. A pattern is either pattern-mask or pattern-request.

A pattern-mask is for selecting properties of a given a class or individual. It needs when a subset of properties is used, and the pattern includes only those properties. Then this pattern is applied to the given class or individual, e.g. for modest updating the properties. For example, let us update only the last name of "Aino" (see the ontology in Figure 4).

```
individual_t *aino_p = new_individual(CLASS_WOMAN);
set_property(aino_p, PROPERTY_LNAME, NULL);
ss_update_by_pattern(aino, aino_p);
```

As a result, only the last name value is transferred to smart space. Compared with `ss_update_property()` the benefit becomes obvious when KP needs to update several properties at once or it can form the property subset only in run-time. The same scheme works for population to transfer data modestly from smart space.

A pattern-request is for compact definition of search queries to smart space. A pattern is filled with those properties and values that characterize the individual to find. For example, let us find all men whose first name is "Timo" and wife's first name is "Aino".

```
individual_t *timo_p =
    new_individual(CLASS_MAN);
individual_t *aino_p =
    new_individual(CLASS_WOMAN);

set_property(timo_p, PROPERTY_FNAME, "Timo");
set_property(aino_p, PROPERTY_FNAME, "Aino");
set_property(timo_p, PROPERTY_HAS_WIFE, aino_p);

timo_list = ss_get_individuals_by_pattern(timo_p);
```

In this example, two patterns ("Timo" and "Aino") and two properties (datatype "fname" and object "has_wife") form a subgraph. The SmartSlog library matches the subgraph to the smart space content. As a result, a list of available individuals is returned. Currently, searching leads to iterative triple exchange and matching at the local side. In future, it can be implemented on the top of SPARQL [20], and the most processing moves to the SIB side.

V. Implementation Optimizations

SmartSlog is primarily oriented to low-performance devices [21] and uses a limited subset of ANSI C [22]. 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.

Each ontology entity is implemented as a C structure 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.

SmartSlog provide constants that limits the number of entities, hence the developer can control the code size. Furthermore, one KP often needs only a subset of them (see our example in Section VI). SmartSlog allows the developer to 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.

Inserting and receiving individuals to and from smart spaces lead to transferring a lot of triples. The network traffic can be reduced by transfer a subset only. SmartSlog API allows the KP developer to explicitly select (using patterns, see Section IV) what properties to use in an operation. That is, even if an individual in the smart space has dozens of properties, KP can populate only few of them (the others are unused this time). Moreover as we discussed above, KP can also deselect totally unneeded properties from its code.

As a result, KP works locally with a subset of properties required by KP semantics at current time instance. There is no need to load/save all properties from/to the smart space.

Smart-M3 CodeGen keeps a triple store—a local cache of smart space content. For large ontology it is expensive. In contrast, SmartSlog does not intend to store any triple for long time. Ontology entities are stored in own structures. When a triple is needed it is created and processed. Then the memory is freed immediately after the usage.

SmartSlog supports both types of subscriptions: synchronous and asynchronous. The latter case requires threading. SmartSlog uses POSIX threads [23] available on many embedded systems [21]. Nevertheless, SmartSlog allows switching the asynchronous subscription off if the target device has no thread support.

VI. Use Case Example

In this section, we show how SmartSlog can be used for constructing a simple Smart-M3 application. In spite of the simplicity, the example illustrates such SmartSlog features as patterns and subscriptions (synchronous and asynchronous). Both datatype and object properties are used.

Let Ericsson's family consist of Timo (husband) and Aino (wife). Timo likes drinking beer outside home. Aino has to control Timo's drinking via monitoring the amount of beer he has drunk already. If the amount is exceeding a certain bound (e.g., `MAX_LITRES_VALUE=3`) she notifies Timo by SMS that it's good time to come back to home.

The ontology for such personal human data was shown in Figure 4 above. When Timo starts drinking he associates his object property "drinks" with class "Beer". Then Timo keeps his drink counter "number_of_drinks" in smart space and regularly updates it. Aino can subscribe to this counter.

For messaging, the family uses the ontology shown in Figure 5. Aino sends SMS to notify Timo via smart space.

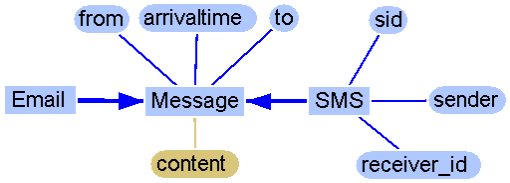

Figure 5. Ontology for messaging

Timo subscribes for SMS and checks each SMS he received for who sent it (by phone number). Hence Timo recognizes a notification SMS from his wife.

Given these two ontology files, SmartSlog generator produces files `drinkers.{c, h}`. Since the ontology includes more details than needed for this application, excessive classes and properties can be disabled in the final code by compiler preprocessor directives.

The KP code for Timo can be constructed with SmartSlog using the following scheme.

1. Create Timo, set his properties, and insert the individual to the smart space.

```
individual_t *timo = new_individual(CLASS_MAN);
set_property(timo,PROPERTY_FNAME, "Timo");
     . . .
ss_insert_individual(timo);
```

2. Timo keeps his counter in the smart space.

```
individual_t *beer = new_individual(CLASS_BEER);
ss_set_property(timo, PROPERTY_DRINKS, beer);
```

3. Timo subscribes to SMS from Aino: creating an individual for SMS and filling the subscribe container. Then asynchronous (parameter “true”) subscription starts.

```
individual_t *sms = new_individual(CLASS_SMS);
add_data_to_list(subscribed_prop_list,
     PROPERTY_FROM);
add_data_to_list(subscribed_prop_list,
     PROPERTY_TO);

subscription_container_t *container=
     new_subscription_container();
add_individual_to_subscribe(container,
     sms, subscribed_prop_list);

ss_subscribe_container(container, true);
```

4. Timo drinks, updates the counter, and checks SMS.

```
while(sms_notify(sms)) {
    amount += drink(timo);
    ss_update_property(timo,
         PROPERTY_NUMBER_OF_DRINKS, amount);
}
```

Similarly, the KP code for Aino is constructed as follows.

1. Aino searches Timo in the smart space by pattern.

```
individual_t *wife = new_individual(CLASS_WOMAN);
set_property(wife, PROPERTY_LNAME, "Ericsson");
set_property(wife, PROPERTY_FNAME, "Aino");

individual_t *timo = new_individual(CLASS_MAN);
set_property(timo, PROPERTY_FNAME, "Timo");
set_property(timo, PROPERTY_HAS_WIFE, wife);
     . . .

list = ss_get_individuals_by_pattern(timo);
```

2. Synchronous (parameter “false”) subscription waits for Timo is starting to drink.

```
subscription_container_t *container=
     new_subscription_container();
add_individual_to_subscribe(container, timo,
                            properties);
ss_subscribe_container(container, false)

property_t *drinks = get_property(timo,
                        PROPERTY_DRINKS);
if (drinks==NULL) wait_subscribe(container);
```

3. Monitoring Timo’s counter and checking the limit. Synchronous subscription is similar to the above.

```
/* Subscribing for Timo’s counter */
   . . .

while(1) {
    amount = get_property(timo,
              PROPERTY_NUMBER_OF_DRINKS);
    if (amount >= MAX_LITRES_VALUE) {
        /* Send SMS to Timo */
        break;
    }
    wait_subscribe(container_counter);
}
```

4. Create an individual for SMS and insert it to the smart space. Properties “to” and “from” are required.

```
individual_t *sms=new_individual(CLASS_SMS);
set_property(sms, PROPERTY_TO,
                  TIMO_PHONE_NUMBER);
set_property(sms, PROPERTY_FROM,
                  WIFE_PHONE_NUMBER);
ss_insert_individual(sms);
```

VII. Conclusion and Future Work

The addressed area of high-level ontology library generation for low-performance devices is very important. The realization of the ubiquitous computing vision will by definition include a lot of small devices around us. Allowing these small devices to easily share information with other devices and architectures, large or small, will be very important.

In this paper we described SmartSlog—a tool that supports efficient programming such devices for participating in smart space applications. The resulting code is compact due to high-level ontology style, portable due to adhering to ANSI C and POSIX standards, modest and optimizable to device capacity due to the design. We believe that SmartSlog will become an important element of the Smart-M3 platform.

The paper presented our work-in-progress. The future work includes more optimization depending on the needs of a concrete KP. For example, ontology metainformation allows defining what types of embedded devices can use a certain part of ontology. It leads to implementing various ontology manipulations that utilize metainformation on

versioning, namespaces, and other differentiation characteristics. Another important direction of our future work is optimization of the SIB ↔ KP communication. For example, a part of triple-based processing can be moved to the SIB side using SPARQL query language; its support will appear in Smart-M3 soon.

ACKNOWLEDGMENT

Authors would like to thank Finnish-Russian University Cooperation in Telecommunications (FRUCT) program for the provided support and R&D infrastructure. The special thanks to Nokia university collaboration program for providing publication grant and all FRUCT experts for commenting and reviewing the project. We would also like to thank Vesa Luukkala and Ronald Brown from Nokia Research Center for providing feedback and guidance during the construction of the SmartSlog tool.

REFERENCES

[1] I. Oliver, J. Honkola, and J. Ziegler, "Dynamic, localised space based semantic webs," in *Proc. IADIS Int'l Conf. WWW/Internet 2008*. IADIS Press, Oct. 2008, pp. 426–431.

[2] I. Oliver, "Information spaces as a basis for personalising the semantic web," in *Proc. 11th Int'l Conf. Enterprise Information Systems (ICEIS 2009)*, vol. SAIC, May 2009, pp. 179–184.

[3] D. Fensel, W. Wahlster, H. Lieberman, and J. Hendler, Eds., *Spinning the semantic web : bringing the World Wide Web to its full potential*. The MIT Press, 2005.

[4] R. Baldoni, M. Contenti, and A. Virgillito, "The evolution of publish/subscribe communication systems," in *Future Directions in Distributed Computing*, ser. Lecture Notes in Computer Science, vol. 2584. Springer, 2003, pp. 137–141.

[5] I. Oliver, E. Nuutila, and S. Törmä, "Context gathering in meetings: Business processes meet the agents and the semantic web," in *The 4th Int'l Workshop on Technologies for Context-Aware Business Process Management (TCoB 2009) within Proc. Joint Workshop on Advanced Technologies and Techniques for Enterprise Information Systems*. INSTICC Press, May 2009.

[6] A. Smirnov, A. Kashnevik, N. Shilov, I. Oliver, S. Balandin, and S. Boldyrev, "Anonymous agent coordination in smart spaces: State-of-the-art," in *Proc. 9th Int'l Conf. Smart Spaces and Next Generation Wired/Wireless Networking (NEW2AN'09) and 2nd Conf. Smart Spaces (ruSMART'09)*, ser. Lecture Notes in Computer Science, vol. 5764. Springer-Verlag, 2009, pp. 42–51.

[7] K. Främling, I. Oliver, J. Honkola, and J. Nyman, "Smart spaces for ubiquitously smart buildings," in *Proc. 3rd Int'l Conf. Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM 2009)*. IEEE Computer Society, 2009, pp. 295–300.

[8] J. Honkola, H. Laine, R. Brown, and I. Oliver, "Cross-domain interoperability: A case study," in *Proc. 9th Int'l Conf. Smart Spaces and Next Generation Wired/Wireless Networking (NEW2AN'09) and 2nd Conf. Smart Spaces (ruSMART'09)*, ser. Lecture Notes in Computer Science, vol. 5764. Springer-Verlag, 2009, pp. 22–31.

[9] S. Balandin, I. Oliver, and S. Boldyrev, "Distributed architecture of a professional social network on top of M3 smart space solution made in PCs and mobile devices friendly manner," in *Proc. 3rd Int'l Conf. Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM 2009)*. IEEE Computer Society, 2009, pp. 318–323.

[10] J. Honkola, H. Laine, R. Brown, and O. Tyrkkö, "Smart-M3 information sharing platform," in *The 1st Int'l Workshop on Semantic Interoperability for Smart Spaces (SISS 2010) in conjunction with IEEE ISCC 2010*, Jun. 2010.

[11] "Download Smart-M3 software for free at SourceForge.net," Release 0.9.4beta, May 2010. [Online]. Available: http://sourceforge.net/projects/smart-m3/

[12] "Download SmartSlog software for free at SourceForge.net," Release 0.22, Apr. 2010. [Online]. Available: http://sourceforge.net/projects/smartslog/

[13] P. Liuha, A. Lappeteläinen, and J.-P. Soininen, "Smart objects for intelligent applications - first results made open," *ARTEMIS Magazine*, no. 5, pp. 27–29, Oct. 2009.

[14] "RDFAlchemy," Jul. 2010. [Online]. Available: http://www.openvest.com/trac/wiki/RDFAlchemy

[15] "Datagraph's spira at github," Version 0.0.5, Jun. 2010. [Online]. Available: http://github.com/datagraph/spira

[16] "Jena – a semantic web framework for java," Jul. 2010. [Online]. Available: http://jena.sourceforge.net/

[17] "Download KPI_low software for free at SourceForge.net," Jun. 2010. [Online]. Available: http://sourceforge.net/projects/kpilow/

[18] T. Littlefair, "CCCC — C and C++ code counter," May 2010. [Online]. Available: http://cccc.sourceforge.net/

[19] S. Krivov, R. Williams, and F. Villa, "GrOWL: A tool for visualization and editing of OWL ontologies," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, no. 2, pp. 54–57, 2007.

[20] 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/

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

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

[23] "Standard for information technology — portable operating system interface (POSIX)," Tech. Rep. 1003.1-2001/Cor 2-2004, 2004.