FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Semantic XVSM

Design and Implementation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Lukas Klausner
Matrikelnummer 0425315

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: A.o.Univ.-Prof. Dr. Dipl.-Ing. eva Kühn

Wien, 28.04.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Semantic XVSM

Design and Implementation

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Lukas Klausner
Registration Number 0425315

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: A.o.Univ.-Prof. Dr. Dipl.-Ing. eva Kühn

Vienna, 28.04.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Lukas Klausner
Friedrich Schiller-Straße 101/3/1, 2340 Mödling

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Smithfield vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

The writing of this thesis took me three years. I want to give my special thanks to the persons who helped me make it a reality.

My wife and my son gave me a lot of energy, and suffered my absences. Jack Wierzchowski corrected my English, and helped me to make this thesis more readable. My parents staunchly supported, in more than one way, my ”little research project” . I thank my advisor, eva Kühn, for the exciting topic and for her patience.

Abstract

The coordination of distributed software agents can be a complex task, especially if the agents are developed by different parties. Space-based computing—a distributed shared memory concept for data-driven communication—tries to simplify the development of such application scenarios by decoupling the communication from time and location. Semantic Web enables the development of domain models and knowledge with platform-independent web technologies. To improve the interoperability in heterogeneous systems, Semantic XVSM—the framework developed in this thesis—enables the use of semantic models and knowledge inside the spaces-based computing framework XVSM. The use of Semantic Web technologies in XVSM offers new and powerful query capabilities for selecting communications items and their enrichment with implicit information.

With Semantic XVSM it became possible to formulate the coordination logic with platform-independent, standardized languages, and to manage this logic at run time by using standard operations of XVSM. The coordination logic describes the routing of data in XVSM.

In the past, certain projects already combined Semantic Web technologies with space-based computing to a semantic tuple space. This thesis concentrates on the development of a solid foundation for further frameworks, and stresses the exclusive use of W3C standards. Many features of these standards can be used unrestricted. The latest versions of the Semantic Web technologies offer new design options for a semantic tuple space framework.

In this thesis, the design of Semantic XVSM and its reference implementation, Semantic MozartSpaces, are presented. Some of the challenges to solve were the transaction handling between XVSM and a triplestore, and the full conformity of the semantic coordinator with the coordinator staging concept of XVSM. A use case application that solves a problem in a multi agent system is presented. The differences of Semantic XVSM to native XVSM, to previous projects, and to triplestore are evaluated. Some application scenarios are also described.

Kurzfassung

Die Koordination von verteilten Softwareagenten kann eine komplexe Aufgabe darstellen, speziell wenn diese Agenten von verschiedenen Organisationen entwickelt wurden. Space-based-Computing – ein Konzept für datengetriebene Kommunikation mittels verteiltem gemeinsam benutzten Speicher – versucht die Entwicklung von solchen Applikationsszenarien durch Entkopplung der Kommunikation von Zeit und Ort zu vereinfachen. Semantic-Web ermöglicht die Entwicklung von Domänenmodellen und Wissensbasen mit plattformunabhängigen Web-Technologien. Zur Verbesserung der Interoperabilität in heterogenen Systemen ermöglicht Semantic XVSM – das in dieser Diplomarbeit entwickelte Framework – den Gebrauch von semantischen Modellen und Wissen innerhalb des Space-based-Computing-Frameworks XVSM. Der Gebrauch von Semantic-Web-Technologien in XVSM erlaubt die Selektion von Kommunikationsobjekten mittels mächtigen Abfragefähigkeiten und die Anreicherung dieser Objekte mit impliziter Information.

Mit Semantic XVSM wurde es möglich die Koordinationslogik mit plattformunabhängigen, standardisierten Sprachen zu formulieren und diese Logik zur Laufzeit unter Benutzung von XVSM-Standardoperationen zu verwalten. Die Koordinationslogik beschreibt die Verteilung der Objekte in XVSM.

Einige Projekte kombinierten bereits Semantic-Web-Technologien mit Space-based Computing zu einem semantischen Tuple-Space. Diese Diplomarbeit konzentriert sich auf die Entwicklung einer soliden Basis für zukünftige Frameworks und verwendet dazu ausschließlich W3C-Standards. Viele Funktionen dieser Standards können uneingeschränkt verwendet werden. Die neuesten Versionen der Semantic-Web-Technologien bieten neue Designoptionen für ein semantisches Tuple-Spaces-Framework.

In dieser Diplomarbeit wird das Design von Semantic XVSM und dessen Referenzimplementierung, Semantic MozartSpaces, vorgestellt. Eine Lösung für die Transaktionsabwicklung zwischen XVSM und einem Triplestore und die völlige Konformität des semantischen Koordinators mit dem Stufenkonzepts von XVSM, welches die Aneinanderreihung mehrerer Koordinatoren innerhalb einer Abfrage unterstützt, waren unter anderem eine Herausforderung. Eine Applikation zur Lösung eines Problems in einem Multiagentensystem wird als Anwendungsbeispiel vorgestellt. Die Unterschiede zwischen Semantic XVSM und dem nativen XVSM, den vorangegangen Projekten und einem Triplestore werden evaluiert. Einige Applikationsszenarien werden präsentiert.

Contents

CHAPTER 1

Introduction

In this thesis, Semantic XVSM and its reference implementation, Semantic MozartSpaces, are introduced. The accompanying source code is published under the open source license AGPL 3 [3].

Semantic Web technologies have a great potential to enrich space-based computing. In this thesis, a design for the combination of these two technologies is presented and arguments made for the chosen design. The focus of this thesis lies on the W3C standards, including RDF [MM04], OWL [BFH+12] and SPARQL [HS13].

Space-based computing is a powerful concept for coordination middleware. It follows the blackboard metaphor. Software agents can post data as tuples, and can retrieve tuples that match a certain pattern. Agents can communicate with each other without being reachable at the same time, and without knowing each other, especially their location. The origin of space-based computing is the tuple spaces paradigm that was invented by Carriero and Gelernter [CG89]. The XVSM specification [Cra10] extends and improves this paradigm.

In this thesis, XVSM is extended to optionally use SPARQL for selecting tuples. Tuples are enriched with their inferred data by using an OWL reasoner. The ontology used for reasoning is managed by the user. The consistency of new tuples is checked against such an ontology. Invalid tuples are rejected. The complete state of a Semantic XVSM runtime is persisted in a triplestore.

Out of scope of this thesis are:

- a consistency check of the complete state of a space,
- a subscription mechanisms to get notified when a specific space state is reached,
- concepts for mediation, or mapping between different data models,
- an integration with a P2P framework,
- an automatic entry distribution to spaces that are responsible for their data, and
- replication mechanisms.

In this thesis, nearly every possible combination of Semantic Web technologies and space-based computing was considered. Several ways of integrating them were tried out. The different options and possibilities were compared. The design was radically changed several times. These iterations were done until all parts of the solution fit together, and no better design options could be found. Three functional prototypes had been build before this thesis was written and the final version of Semantic MozartSpaces was developed.

In a cooperation with Domagoj Drenjanac, a PHD student at the Telecommunications Research Center Vienna (FTW), two scientific papers [DKKT13] and [DKKTon] were written. They described a use case in a multi-agent system for solving tasks in the field of agriculture. The presented solution for this use case applied Semantic XVSM. This cooperation enabled a discussion and additional review of the concepts presented in this thesis.

At the beginning of this thesis, background information about XVSM and the Semantic Web technologies is presented. This information is essential to understand the rest of the thesis. In chapter 4, existing semantic tuple spaces projects are reviewed and compared. Chapter 5 includes a detailed list of the requirements and the description of the design of Semantic XVSM. This description contains the argumentation for the design decisions. Semantic MozartSpaces—the reference implementation of Semantic XVSM—is described in chapter 6. The textual explanations are supported by UML diagrams. The user guide (chapter 7) describes the use of Semantic MozartSpaces for application developers. A use case application that solves a task allocation problem in a multi agent system is presented in chapter 8. The evaluation of Semantic XVSM (chapter 9) includes a comparison of Semantic XVSM with XVSM, other semantic tuple spaces projects, and plain triplestores. Benchmarks illustrate the practical feasibility of Semantic MozartSpaces. The results of this thesis are summarized in chapter 11. Most of the ideas and suggestions that should be addressed in future works are presented in the descriptions of the design and its implementation. Additional remarks can also be found in chapter 10.

CHAPTER 2

Tuple Spaces and XVSM

This chapter provides background information about tuple spaces and the specification XVSM, which is extended by this thesis. XVSM specifies a concrete, improved and language independent tuple space architecture and has MozartSpaces as the reference implementation written in Java.

2.1 Tuple Spaces

Tuple spaces, a framework for parallel programming based on the blackboard paradigm, was invented by Gelernter and Carriero [CG89]. The blackboard paradigm is quite simple: somebody can post a sheet (tuple) on the blackboard, everybody can read it, and someone can take it down. The advantage of this paradigm is that people or computers can coordinate and communicate with themselves without knowing each other, and do not have to be reachable accessible at the same time. Furthermore, sharing information is very easy. In tuple spaces, tuples are written into a space (blackboard) and can be read or taken (a destructive read). Optionally, read operations run as long as no matching tuple is present in the space (a blocking manner). A tuple is a nested, ordered list of literals. Different tuples with the same value are allowed in a tuple space at the same time. For each read operation, a template with the same structure as the requested tuple has to be declared. The fields in such a template include either the same data as the tuple, or a placeholder for any data.

Typically, an application consists of several spaces, which are hosted by their participants. This is the reason why tuple spaces are written in plural. Tuple spaces frameworks are designed to be used in peer-to-peer (P2P) scenarios. In most applications, however, space instances are only hosted by dedicated servers, which are sometimes connected by a P2P network.

A common application scenario is a distributed system, which stores its state in one shared tuple space. Participants write their states into the space to share them with each other. Thus, for solving coordination problems, only one read instead of many distributed polls is necessary. In tuple spaces, the most common pattern is the master worker pattern—similar to map-reduce—, where a master produces jobs by writing new tuples into the space, and several workers take out

one job after another, process them and write the result back into the space. Such a job can involve doing some calculations, but also tasks in the real world, like cooking a pizza. The advantage of the master worker pattern is that by adding more workers an application can scale nearly linearly.

2.2 XVSM Specification

XVSM (eXtensible Virtual Shared Memory) [CKS09] [Cra10] provides extensions and adaptions to the original tuple spaces. A runtime instance of XVSM is called XVSM *core*. Instead of hosting one space, the core is structured into several XVSM *containers*. The containers own exclusively their *entries*—the XVSM equivalent of tuples. But in contrast to a tuple, the fields in the nested list of an entry are labeled and have no special order. In other words, the data structure of an entry is a recursive X-tree, which is either a sequence or a multiset of labeled X-trees, or a literal. Unique IDs (called labels) are assigned to each entry for distinguishing them internally in the core. These IDs are, for example, needed for handling transactional locks. Moreover, a container can be referenced by a URL, which is a combination of the core's URL and its name.

For each operation (read, take, write) a timeout can be declared. Thus, a flexible blocking behavior is possible. Write and remove operations trigger a reschedule of all open blocking read operations. A user-defined lease (expiry) for an entry is planned, but is not yet implemented.

The core API (*CAPI*) is the interface for users to create new containers, write or read entries, and much more.

XVSM can be extended by, for example, writing new aspects or new custom coordinators. There exist extensions for automatic replication [Hir13] and persistency [Zar12].

Transactions

To handle and synchronize concurrent access, XVSM includes a transaction manager with a transaction support. The isolation happens on the entry level and all entries are encapsulated logical entities with no direct dependency among them. XVSM uses a pessimistic isolation locking; exceptions are thrown as soon as a conflict occurs. In order to reschedule single user operations, they are encapsulated in a sub-transaction. In such a case, a rollback of the whole user transaction can be prevented.

There are two isolation levels supported by XVSM: read-committed and repeatable-read. While the former allows only the entries deleted in another transaction to be read, repeatable-read blocks them. Furthermore, repeatable-read sets read-locks, which prevent read entries from deletion in other transactions. But even repeatable-read guarantees only that the read entries are still available at commit. In the case of newly committed entries between two executions of the same read operation, the second read can have more results or even another result.

An outstanding feature of XVSM is the timeout for transactions. In the master worker pattern, a job processing can be automatically rollbacked after a specific amount of time, which is needed, for example, in the case of a crashed worker.

Each XVSM core has its own separate isolation manager. To get support for distributed transactions, Brückl [Brü13] designed and implemented an extension for XVSM.

Coordinator

XVSM offers a flexible concept for selecting entries. Therefore, different coordinators can be attached to a container. While processing a read operation, the selector of a coordinator is called and decides the order and the subset of entries to be returned to the user, who can declare selector-specific parameters and the size (*count*) of the result set. Two special count constants are predefined: *COUNT_MAX* tries to get as many entries as possible, and *COUNT_ALL* returns all entries, or throws an exception in the case any of the selected entries is blocked by the isolation manager.

Coordinator specific metadata (*coordination data*) can be attached to an entry. These data are stored in a branch of the labeled X-tree of their entry, and are no longer changeable as soon as the write operation is finished. The coordinator can request the user to define these metadata (e.g. a label or a key). Beyond coordination of data on the entry level, a coordinator can have an optimized index structure, which is also called "coordination data", and is attached as an X-tree to the container's branch in the XVSM meta model. To be able to manage its own internal data structure, a coordinator is informed about all written and removed entries.

At container creation, the obligation of each coordinator is declared. Either a coordinator manages all entries of a container (*obligatory*), or manages only explicit user-specified entries (*optional*). Table 2.1 gives an overview of the predefined XVSM coordinators. For example, the FiFo (first in first out) coordinator selects first the entries, that were written first. The key coordinator for each entry manages a unique key. But custom containers are also supported.

Coordinator	*Coordination data*	*Selection parameters*	*Coordination law*
any	-	count	arbitrary selection
key	key	key	selection via unique key
label	label(s)	label, count	selection via non-unique label
fifo	-	count	selection in FIFO order
type	-	type, count	selection via data type of entry
query	-	query, count	SQL-like query on entry fields with comparisons, sortings, etc. [CKS09]

Table 2.1: Predefined XVSM coordinators [CDJ+13]

For a read operation, not just one, but a sequence of selectors can be defined. The XVSM runtime processes them in a *selector chain*, where the result of one selector is piped as an input to the next selector. Because sometimes a selected entry of one selector is filtered out by a follower, entries are not transactionally locked until they are finally selected. For optimization, a selector can choose to get the entries of its predecessor either all at once, or by iteration.

Aspect

For every operation of the CAPI, pre- and post-aspects can be defined. Such aspects can block entries from reading or writing, can attach or modify entries, or just log events. Because an aspect

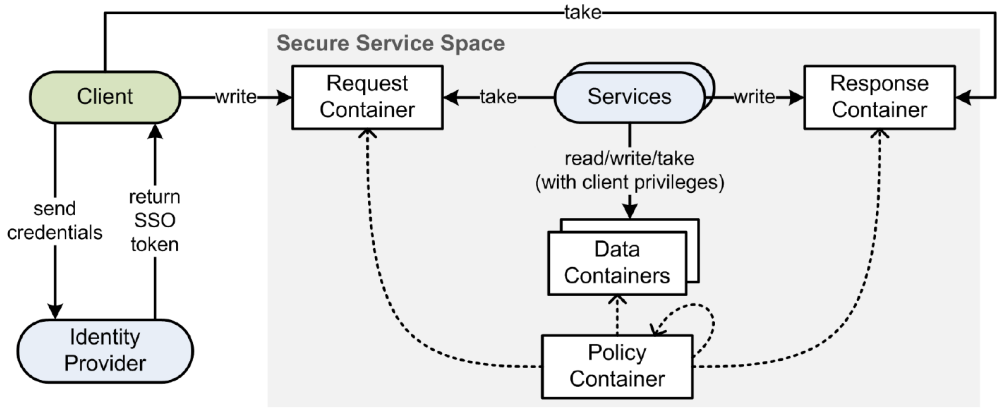

Figure 2.1: Secure Service Space architecture [CDJ+13]

can also call all CAPI operations, it can do much more.

One example of a predefined aspect is the notification manager of MozartSpaces, which notifies all subscribers about every written or read entry.

Access Control

The coordination-based access control model [CDJ+13] helps to secure XVSM in a flexible way. Security restriction rules are defined with coordinators. Therefore, access control rules on the entry level are expressible. Furthermore, because the same concept is used for selecting entries, the additional overhead for security rules is small and the available coordination logic can be used. Figure 2.1 gives an overview of the access control architecture. Before accessing the secured XVSM core, the client has to get a single sign-on (SSO) token from an identity provider.

At the end of a write operation, when the entries with their coordination data are already inserted into the container, a read operation tries to find entries that do not fulfill the security rules. If a forbidden entry exists, the sub-transaction of the write operation will be aborted.

In the case of a read or take operation, the selected entries are checked if they can also be read with a selector that includes the security rules. The selected entries that could not be read are filtered out of the result set. If such an entry is mandatory for a selector, the sub-transaction of the read operation will be aborted.

2.3 MozartSpaces

MozartSpaces [Bar10] [Dön11] is the reference implementation of XVSM written in Java and is published [14] under the AGPL 3 license [3]. The version 2.3, which is still under development, was used for this thesis.

The value of an entry is represented by an immutable, serializable Java object. For write operations on client-side, an entry value is encapsulated with its coordination data in an instance of the class `org.mozartspaces.core.Entry`. Then, on server-

side, for each entry an ID is generated, and its value is put into a new instance of `org.mozartspaces.capi3.javanative.operation.NativeEntry`. The coordination data are managed separately by their corresponding coordinators. A read operation returns to the user a list of serializable objects (entry values). The coordination data are not accessible on client-side. Each container has an ID, which is generated by the XVSM. The optional name of a container enables other users to look up the container.

Everything already written about XVSM is also valid for MozartSpaces. The following subsections present selected features of MozartSpaces.

Transaction Log Items

To trigger custom events at commit or rollback, a log-item (`org.mozartspaces.capi3.LogItem`) is added to a transaction, or to a sub-transaction. Log-items are mainly used in the persistence layer for committing or aborting database transactions. The log-item will be called when a transaction is committed or rollbacked. In the case of a sub-transaction, the log-item will be also called, when the parent-transaction is committed or rollbacked.

Custom log-items are processed after all log-items of the MozartSpaces core. In the case of a rollback of a write request, all coordinators remove the written entries of the transaction in the function `unregisterEntry` before the custom log-items are executed. Log-items are not allowed to throw any exceptions.

Request Context

Each user request has its own request context. It can be used by the user, or by the components of MozartSpaces to forward information in the same request to other components. Apart from other components, aspects and coordinators can also access the request context. For example, a pre-aspect can hand over some data to the post-aspect of the same request. A request context exists for every user operation of CAPI. The request context is implemented as a hash map with strings as keys and any object as a value.

Persistency

The persistence API [Zar12] of MozartSpaces provides the capabilities to restore data, and to have containers, that do not fit into the memory.

The only offered data structure is a `StoredMap`, which is similar to the Java map interface, but throws `PersistenceExceptions`. As a map key `long`, `String` and `NativeEntry` are accepted. All kinds of objects are allowed as map values. For storing in the database, they are converted to byte arrays by the Java serializer.

The read operations of the `StoredMap` interface return all matches independent from any transaction. Therefore, a client of the map, a coordinator implementation has to use the accessibility checker of MozartSpaces to filter out all invisible entries for the transaction, and to make an access control check.

Beside four default persistence profiles, custom profiles are also possible. The default profiles are listed in an ascending order sorted by their speed. First, `TRANSACTIONAL_SYNC_BERKE-`

LEY immediately initiates file synchronization after each change. Second, TRANSACTIONAL_-BERKELEY writes all changes to the disk after each change, but without file synchronization. Third, LAZY_BERKELEY has a buffer for changes, and writes to the disk asynchronously. Last, IN_MEMORY holds all data in the memory only. At the start of a MozartSpaces core, one persistence profile can be declared in the configuration, which is used for all containers except the ones which have the force in-memory flag set.

To offer the coordinators the possibility to rebuild in-memory indexing structures (coordination data on the container level), the persistence API extends the coordinator interface with a restore method.

To save memory and to increase performance, lazy entries were introduced. They hold their values as weak references; thus the values can be removed by the garbage collector. The value of a lazy entry is loaded on-demand from the database. Thus in a selector chain, for example, only the values of the finally returned entries are read from the database. An entry is made lazy before it is inserted into a container.

Additionally, for better read performance, a cache for StoredMap values—usually coordination data— can be used. Hence, the values are not loaded from the database every time.

A persistence back-end profile does not deal with sub-transactions. In the case of a write operation, the rollback of its sub-transaction will trigger the remove of all written data inside such a sub-transaction. The remove operation of a StoredMap is called while a take or delete request is committed. This stipulates that all coordinator implementations remove their coordination data only inside their unregister function.

MozartSpaces transactions are neither restored nor persisted. Thus, after a system crash all uncommitted data are lost.

CHAPTER 3

Semantic Web Technologies

The main idea of the Semantic Web is to enable a way to publish machine-understandable information on the Web. The aim is to create a knowledge network with authors all over the world. Technologies to search for knowledge and to infer new knowledge by reasoning are provided. Objects get a worldwide unique ID; thus anyone can publish data about a special object.

The focus of Semantic Web lies on distributed knowledge management, but, as shown by this thesis, its technologies can also improve communication and coordination systems. For example, coordination in heterogeneous environments can benefit greatly from standardized domain models (vocabularies) with specified semantic meaning, and the visibility of implicit data.

This chapter gives an introduction to the parts of the Semantic Web technology that are relevant for this thesis: W3C standards RDF, Turtle, SPARQL, and OWL.

3.1 RDF

RDF [MM04] is a language to describe resources and is used as the core representation for all data in the Semantic Web. A resource can be a person, a real world object like a house, or even a virtual object like a website. Each resource is uniquely identified by a URI reference, which is a small extension of URI (Uniform Resource Identifiers) [BLFM98]. Any kind of data can be expressed with RDF. The description of one resource can be spread over several servers. It enables people around the world to build together a knowledge base.

RDF data are structured in graphs, where each resource represents a node, which is connected with other resources by the labeled directed edges, which describe relations between two resources. For abbreviation, RDF supports not only absolute (full) URIs, but also relative URIs, with XML namespaces as a prefix. For example, the URI `dc:creator` will be equivalent to `http://purl.org/dc/elements/1.1/creator`, if the namespace `dc` represents `http://purl.org/dc/elements/1.1/`. Figure 3.1 describes this thesis in a simple RDF graph. This example graph purposefully uses several vocabularies, because best practice states, that in order to increase the re-usability of a data model, classes or properties of an existing vocabulary

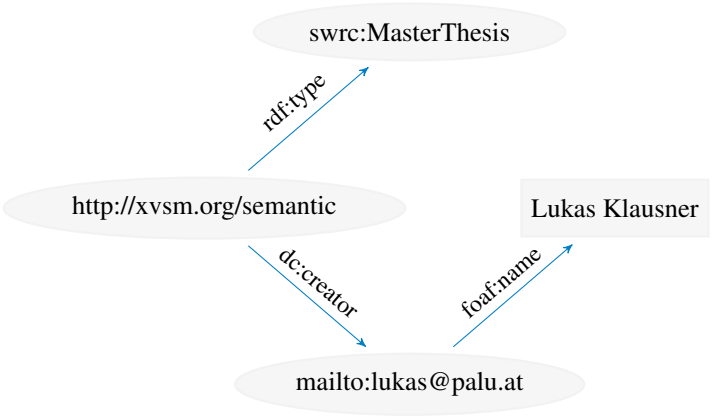

Figure 3.1: RDF graph describing this thesis

should be treated preferentially. A mail address is often used to identify a person. ”mailto” is a URI scheme as ”http”.

Triples are used to formulate RDF data textually. A triple is the counterpart of an edge in an RDF graph, and consists of a *subject*, a *predicate*, and an *object*. A predicate labels the relation between two nodes represented by the subject and the object. Subjects and predicates are both URIs. Beside a URI, an object can also be a literal to express, for example, a string or a number. RDF borrows the literal data types from XML Schema [MB04].

A predicate of an RDF triple is sometime called RDF property (e.g. in SPARQL). A Java object can be easily expressed as an RDF resource. Its field allocations are represented by edges pointing to a value. In other words, the subject is the object's resource, the predicates are the labels of its fields (properties) and the objects are the values of the properties. Listing 3.1, which is also referred to in the next chapter, compares a Java class with an RDF graph, which could be an instance of that class.

In some situations, to express collections for example, extra nodes that are used solely for structuring data, are needed. Therefore, RDF introduces anonymous resources—also called *blank nodes*. Instead of a global, unique identifier, such nodes have only temporal identifiers that are only valid inside a document.

RDF Collection

For some data structures RDF Schema [BG04] provides a recommended standard vocabulary. An RDF collection is the equivalent of a list in Java. It has ordered values, where the same value may occur more than once. Furthermore, the end of the list is expressed by `rdf:nil`, which guarantees a read of the complete list. This is necessary, because in the Semantic Web it is not always guaranteed to get all requested triples, and, an application has to be able to deal with incomplete datasets. Figure 3.2 shows a simple list with two numbers.

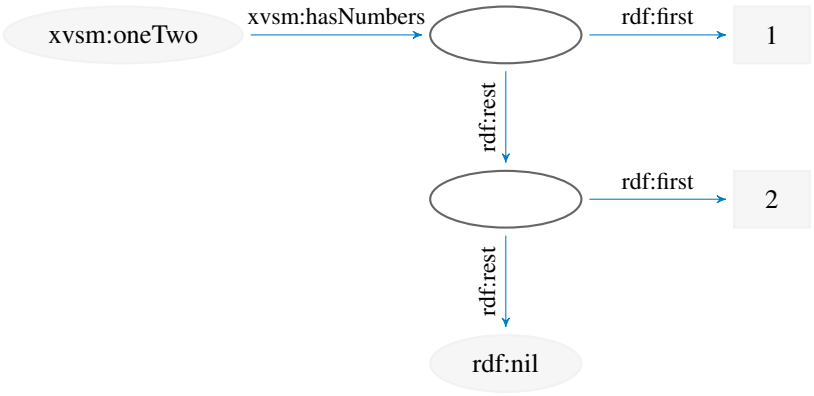

Figure 3.2: RDF collection

3.2 Turtle

Turtle [BBLPC13] defines a terse, textual syntax for storing and exchanging RDF data. In this work, to ensure a consistent presentation, all RDF data are shown in Turtle. SPARQL uses the same syntax for query definitions.

Like in a written sentence, the subject, the predicate and the object of a triple are written in a sequence separated by spaces, and are concluded by a dot (`.`). Absolute (full) URIs are expressed by enclosing them in angle brackets (`< >`). Relative URIs are concatenated with their namespace as a prefix separated by a colon (`:`) and without brackets. The value of a literal is quoted and attached by its XSD type with two preceding carets (`^^`). Finally, a blank node is expressed by `_:` followed by some label, which is only valid inside its document. The RDF graph in figure 3.1 is written in the following turtle sentences.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix swrc: <http://swrc.ontoware.org/ontology#> .
<http://xvsm.org/semantic> rdf:type swrc:MasterThesis .
<http://xvsm.org/semantic> dc:creator <mailto:lukas@palu.at> .
<mailto:lukas@palu.at> foaf:name "Lukas Klausner"^^xsd:string .
```

To enable a compact representation, Turtle offers some abbreviations. For string literals, the declaration of the XSD type is optional. Number literals can be also written without quotes and their XSD type.

A *predicate list*—a set of triples with the same subject—is separated by a semicolon (`;`) and the subject is only written at the beginning.

```
<http://xvsm.org/semantic> rdf:type swrc:MasterThesis ;
                           dc:creator <mailto:lukas@palu.at> .
```

The comma (`,`) is used for *object lists*, which are triples with different objects, but with the same subject and predicate. An object list is equivalent to a Java set. The objects are not ordered and

have to be unique, with the same subject-predicate-combination. The following example assumes that this thesis has two creators.

```
<http://xvsm.org/semantic> dc:creator <mailto:lukas@palu.at> ,
                                      <mailto:hugo@tuwien.ac.at> .
```

The short syntax for an RDF collection is like an object list, but with an enclosing parenthesis (`()`) over all objects.

Particularly important for this thesis is the blank node property list that describes unlabeled *nested blank nodes*. This special representation guarantees that such blank nodes can only be the objects of a single triple. The resource, which links to such a blank node, owns it exclusively. An unlabeled blank node is expressed by square brackets (`[]`). Furthermore, nesting of such blank nodes is possible. In the example of listing 3.1, an anonymous resource of type `ex:Entry` with two properties is shown as an unlabeled nested blank node. The same graph is also written without any Turtle abbreviations, only with RDF triples. For demonstration, a Java class is attached. Objects of this class have the same value as in the presented graph.

3.3 SPARQL

SPARQL is a standardized query [HS13] and update [GPP13] language for RDF data. It also defines an HTTP-based protocol for the communication between a client and a SPARQL service endpoint. SPARQL is powerful and offers a lot of functionalities; hence this chapter can only give a short and incomplete summary.

At the moment, the SPARQL standard does not include any kind of transactional support. One update request can, however, include several operations, which have to be processed by the query engine in one transaction.

Queries are primarily based on template matching, with wild cards used for variables. A template describes a graph and is written in the same syntax as Turtle with some extensions. The query engine looks for every possible combination of triples from the user-defined template. Like SQL, SPARQL provides capabilities for ordering, aggregation, sub-queries and more. Question-marks (`?`) have to precede variable names (e.g. `?id`).

Listing 3.2 shows a query to get a list of IDs of anonymous resources that are the type of `ex:Entry`, have a property `ex:id`, and have a property `ex:value` that is an anonymous resource and has a property `ex:name` with the value ”Hugo”. Assuming that the dataset is the set of triples of the listing 3.1, the result would be the singleton list with ID 1. The graph template inside the `WHERE` clause is exactly the same graph as from the previous listing; only the ID is replaced by the variable ”`id`”, and the property `ex:age` is not necessary. Prefixes are declared similarly to Turtle.

Additionally, to enable more flexible queries, it is possible to declare some sections of a template as *optional*. Beyond template matching, SPARQL offers *filters* for richer restriction capabilities. Among others, a filter constraint can be constructed by logical operators (`&&`, `||`), relational operators (=, `!=`, >), operations (`*`, `/`), and functions for checking the type of variables (`isIRI`, `isBlank`, `isLiteral`).

```
@prefix ex: <http://xvsm.org/semantic/example/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
_:a1 rdf:type ex:Entry .
_:a1 ex:id 1 .
_:a1 ex:value _:b1 .
_:b1 ex:name  "Hugo"^^xsd:string .
_:b1 ex:age  "22"^^xsd:int  .
```

A graph without abbrevations

```
[ rdf:type ex:Entry ;
  ex:id 1 ;
  ex:value [
    ex:name "Hugo"^^xsd:string ;
    ex:age "22"^^xsd:int
  ]
] .
```

The same graph like above, but modeled with blank node property lists

```
class Entry {
  long id = 1;
  Object value = new Object() {
    String name = "Hugo";
    int age = 22;
  };
}
```

An object of this Java class has the same value as the graph.

Listing 3.1: An entry's resource with its value as nested blank node

```
PREFIX ex: <http://xvsm.org/example/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?id
WHERE {
  [ rdf:type ex:Entry ;
    ex:id ?id ;
    ex:value [
      ex:name "Hugo"^^xsd:string ;
    ]
  ] .
}
```

Listing 3.2: SPARQL query

Query Forms

Beyond `SELECT`, SPARQL offers three other query forms, which differ in their result type. The result of a `SELECT` is a table, where each line includes one variable assignment for each matched sub-graph in the dataset. With `CONSTRUCT` a new graph can be generated, a graph segment can be read out, or a graph can be converted into another model. An `ASK` returns true or false depending on the match of some sub-graph in the dataset, and finally, a human can browse big graphs by using `DESCRIBE`. The implementations of `DESCRIBE` differ, because its behavior is not precisely defined in the SPARQL standard. Therefore, this last query form is not suitable for the framework developed in this thesis.

Listing 3.3 demonstrates a `CONSTRUCT` query for reading a branch of a nested blank node list. This query gets only the properties of the first level. Thus the value of `ex:address` is an empty blank node. In the next subsection, a method to get all nested properties recursively is presented.

```
[ rdf:type ex:Entry ;
  ex:id 1 ;
  ex:value [
    ex:name "Hugo"^^xsd:string ;
    ex:address [
      ex:street "1st Street"^^xsd:string
    ]
  ]
] .
```

Dataset

```
CONSTRUCT {
  ?s ?p ?o
}
WHERE {
  ?entry ex:id 1 ;
         ex:value ?s .
  ?s ?p ?o .
}
```

Query

```
[ ex:name "Hugo"^^xsd:string ;
  ex:address []
] .
```

Result includes only properties of value's first level

Listing 3.3: Read the value of an entry with a simple CONSTRUCT query

Property Path

With a property path, two nodes that are connected by a specific path can be found. Such a path describes the allowed predicate combination between these two nodes. A simple path describes the exact sequence of predicates—concatenated with slashes (`/`)—between two nodes. Much more sophisticated paths are also possible, for example, by using the zero-or-more (`*`) or the negation (`!`) expressions.

A property path offers the only way in SPARQL for recursion over different graph levels. This is necessary for reading a whole nested blank node at once. For example, listing 3.4 extends listing 3.3 to get the whole entry value by using a property path. The property path `ex:value/(!rdf:nothing)*` delivers all nodes that are in any way connected to the value's node. The negation of `rdf:nothing` is a workaround to express any predicate. The query from the listing can read too many triples, because it reads also all graphs that are connected to the leafs of the nested blank node. Unfortunately, there is no solution based on recursion for this problem, because property path offers no restriction capability for the nodes included in the path. These nodes must be all blank nodes to read only a nested blank node.

```
CONSTRUCT {
  ?s ?p ?o .
}
WHERE {
  ?entry ex:id 1 ;
         ex:value/(!rdf:nothing)* ?s .
  ?s ?p ?o .
}
```

Extended query

```
[ ex:name "Hugo"^^xsd:string ;
  ex:address [
    ex:street "1st Street"^^xsd:string
]
```

Result (includes the whole nested blank node)

Listing 3.4: Recursive read for nested entry values

Named Graph

To be able to distinguish data from different sources, SPARQL supports named graphs. A named graph is an RDF graph, which is identified by a URI. Named graphs can also be used as additional instrument for data structuring.

Graph-specific sections in the query template can be defined by preceding the keyword `GRAPH` with a named graph's URI. Additionally, the default dataset—the default graph, on which

a query is executed—can be defined by a set of named graph's URIs by using the keyword `FROM`. The query in listing 3.5 gets all triples of the named graph `http://graph-name`.

SPARQL has no support for nested graph names to achieve a graph hierarchy with sub-graphs.

```
CONSTRUCT {
  ?s ?p ?o
}
WHERE {
  GRAPH <http://graph-name> {
    ?s ?p ?o .
  }
}
```

Listing 3.5: Read of all triples of named graph

Federated Query

SPARQL offers a functionality for a distributed join. A specified part of a query can be declared to be executed by a separate SPARQL endpoint—a remote accessible SPARQL service over HTTP. The endpoint sends the result of the sub-query back to the main query engine, which uses the results to solve the query.

With the help of federated queries, not all data required to solve a query have to be stored and managed by the query executing computer. For example, a booking platform gets additional information from DBpedia [9] to present its customer with only these hotels in the city, that were used as the sets of award-winning films. DBpedia offers a huge semantic dataset of Wikipedia's structured data.

3.4 OWL

OWL [BFH+12] is the W3C recommended language for defining ontologies. With OWL axioms, a domain model can be described to infer implicit knowledge and to check its correctness. The creation of new knowledge is done by a reasoner.

RDF Schema [BG04], the predecessor of OWL, is less powerful, and therefore it can be used only for simple domain models, but needs less resources for reasoning. To be downwardly compatible, OWL includes the whole RDF Schema vocabulary. To enable more efficient and less complex reasoners, three subsets of OWL, differing in their expressiveness, are specified as *profiles* [CCG+12].

OWL offers facilities for expressing meaning and semantics of RDF data. Logical relationships are made explicit through OWL class definitions and other formal statements. Among semantics for generalization-hierarchies of RDF properties and classes, OWL offers elements to specify: relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, rich typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes. Beside explicit declarations, an RDF resource derives its types from its relationships (properties).

The members of a relationship become an instance of the classes, which are declared for that relationship. For example, in an ontology about families, a resource will become an instance of the class `:Father`, if it has the property `:isFatherOf`.

In opposite to Java, OWL classes can overlap, and an OWL instance can belong to multiple classes. Furthermore, an OWL class can be created at run time, and a property is a stand-alone entity that can exist without specific classes.

One design goal of OWL was to enable reusability of ontologies. Well-tested ontologies in particular shall be accessible with their URI on the Web. Existing ontologies can be extended by using import expressions in the new ontology. Commonly, an OWL ontology is stored and exchanged as an RDF document, and should use its own unique namespace. To create an ontology, a special editor is advisable. Protégé [17] is such a widely used editor.

Inferencing

The aim of inferencing is to create new data by applying the OWL axioms formulated in an ontology. For example, a resource can become an instance of an additional class. By extending the previous example, a resource, which is an instance of the class `:Father`, will become also an instance of the class `:Man`.

The description of a resource can be enriched in different ways. A resource becomes an instance of additional classes by, for example, `subclass`, `domain`, `range`, `equivalentClass`, or `sameAs` declarations. With `SubObjectPropertyOf`, `EquivalentObjectProperties`, `EquivalentDataProperties` declarations, additional properties with the same value can be added to a resource. Moreover, additional properties for a resource can be inferred by `TransitiveProperty`, `SymmetricProperty`, or some OWL restrictions. Lastly, by using OWL restriction axioms (e.g. `FunctionalProperty`) two resources can become identical (`sameAs`).

Restrictions

With OWL restrictions, the characteristics of the properties of a class instance can be restricted. The type or the occurrence (range) of property's values can be restricted.

Commonly, restriction classes have no declared instances; instead, resources become an instance of a restriction class by reasoning. Therefore, a resource must fulfill the restrictions. The attachment of new properties to resources that are an instance of a restriction class, is also possible. Restrictions can also be used for checking consistency of data.

Listing 3.6 shows a small example of an ontology with a restriction class and the inferred data. The example demonstrates that the reasoning of restriction classes works in both directions. The individual `:pizza1` gets the new property that it has a mozzarella topping, because it is type of `:MozzarellaPizza`. The individual `:pizza2` becomes type of `:MozzarellaPizza`, because it has the property that it has a mozzarella topping.

```
:MozzarellaPizza rdf:type owl:Class ;
                 owl:equivalentClass [rdf:type owl:Restriction ;
                                      owl:onProperty :hasTopping ;
                                      owl:hasValue :mozzerella
                                     ] .
:pizza1 rdf:typ :MozzarellaPizza .
:pizza2 :hasTopping :mozzerella .
```

Ontology

```
:pizza1 :hasTopping :mozzerella .
:pizza2 rdf:typ :MozzarellaPizza .
```

Inferred data

Listing 3.6: Ontology with a restriction class

Consistency

It is possible for an ontology to become inconsistent. The most common case is when a resource becomes an instance of two disjoint classes, or two instances are declared to be equivalent and disjoint at the same time. If an instance of a restriction class cannot fulfill all restrictions, the ontology will also become inconsistent (e.g. a violation of a restricted cardinality of a specified property).

3.5 Triplestore

A triplestore resembles a semantic database. It manages and persists an RDF dataset, which is structured into one default graph and several named graphs. Beside a native API for storing and reading the data, it incorporates a SPARQL engine. Additionally, some triplestores include an OWL reasoner.

The two most popular triplestores that are Java-implemented and are still being developed under an open source license are Apache Jena [8] and Sesame [18]. Both support the latest SPARQL standard and have store back-ends for persistency as well as for in-memory. The isolation managers of both support only one single write transaction, and with workarounds, few short-lived concurrent transactions. The licenses of both Jena (Apache License, version 2.0. [2]) and Sesame (the BSD 3-Clause License [1]) permit their use in MozartSpaces (AGPL). Additionally, Jena includes an open source reasoner for OWL 1 with a moderate performance.

3.6 Reasoner

An OWL reasoner produces inferred data for an ontology and can check its consistency.

Generally, reasoning can be done only over one graph. A dataset with different named graphs is not supported. For processing, most of the reasoners have to load into the memory the complete

dataset and its data structure.

The OWL API [19] is an abstraction framework for the most prominent open source OWL reasoners (ELK [10], FaCT++ [11], JFact [13], HermiT [12] and Pellet [16]).

CHAPTER 4

Semantic Tuple Spaces

This chapter provides an overview and a comparison of the previous projects striving to build a semantic tuple spaces framework. In the literature, several other synonymous terms can be found: triple spaces, semantic spaces, or semantic tuple space computing. Projects in this area strive to combine the Semantic Web technologies with a tuple spaces framework to offer a middleware for solving complex coordination problems in heterogeneous environments.

In this chapter, four most recent projects are presented, and then compared with each other. Core design decisions, especially pertaining to the data model, are compared.

4.1 sTuples

Semantic Tuple Spaces [KLF04] (sTuples) was one of the first projects that attempted to achieve semantic interoperability of tuple spaces in heterogeneous environment by making use of the Semantic Web technologies. Semantic tuples consist of RDF graphs, which are stored in the knowledge base of a reasoner. For each tuple, an own named graph is created in the reasoner. After every insert, the consistency of the knowledge base is checked. The parameter of a read operation is a semantic template. The template language was created by the sTuples project. The query processor iterates over all tuple graphs and compares the similarities with the template. sTuples strives to simplify ubiquitous computing applications (e.g. building automation). In the example presented in [KLF04], a light control is automatized in a room.

4.2 TSC

TSC (Triple Space Computing) project was supported by the Austrian Government. Its overview is given by the paper [FKS+07] and the details are described in several deliverables on the project's website [4].

The data of tuples are RDF graphs, which are stored in separate named graphs into the triplestore. The URI of such a named graph is generated by the server instance (called kernel)

and is returned to the user as the result of write operations. To know which tuples belong to a container, their named graphs are stored in an extra named graph for the container's metadata. This is needed, because a kernel can host several spaces, but only one triplestore per kernel is used.

Because for every written tuple a new ID is generated, different tuples with the same data are allowed. To read a tuple, the user has to pass either the URI of the tuple, or a template describing in the N3QL syntax the requested named graph. N3QL is based on template matching, and is similar to SPARQL, but older and has more limited functionality. The query, which is created from a template, is comparable with the SPARQL query form called `CONSTRUCT`. In addition to the common read operation, which always returns only one tuple, a second, non-destructive read operation called a query is offered. The input of such a query is also a template, but it constructs a new graph by using as a dataset all named graphs in the kernel. Thus, the output is a selection of triples of several named graphs (tuples).

For reading tuples, the URIs of the named graphs that match the user template are resolved from the triplestore, and then their values are retrieved from the corresponding Corso tuples. TSC uses Corso [Küh94] as the tuple spaces framework, and YARS [HD05] as the triplestore framework. Both frameworks cannot run embedded in a Java program and therefore they have to be started separately.

TSC can persist all its data; thus a kernel state can be recovered after a crash or a restart. Because the tuples are stored in Corso as well as in YARS, this doubles the size of the necessary disk storage. A publish-subscribe mechanism is also offered. It informs about newly inserted tuples that match a user-defined template. It is the same mechanism, which is also used for reading. As a use case application, A framework for semantic web services based on the WSMO specification [RKL$^+$05] was implemented.

Transactions

It appears that on the tuple level, TSC offers support for concurrent transactions, but only when using Java Berkley DB as a persistent back-end of YARS. Berkley transactions are mapped to the corresponding Corso transactions, but there are no details given about the concrete mapping. The management (create, commit, abort) of transactions is done via the Corso API. Because an optimistic concurrency control is used, operations are not blocked by locks, but the validation of a transaction before the commit can cause an abort.

TSC++

TSC was further developed (until 2009) in the TSC++ project [7]. Corso was replaced by the peer-to-peer framework JaTX, and instead of YARS, Sesame was used. Furthermore, the transaction support was removed.

4.3 TripCom

TripCom [6] was the largest project pertaining to semantic tuple spaces. Supported by the European Union, TripCom was a collaborative effort of several universities and companies. It was

finished in 2009, after three years of work. The focus was on developing a distributed semantic knowledge-base based on tuple spaces technology.

As part of TripCom, ORDI (Ontology Representation and Data Integration Framework) [5] was developed to abstract the querying and reasoning support. It was built on top of TRREE (Triple Reasoning and Rule Entailment Engine), which uses Sesame. With some restrictions, it can also be used with the triplestore YARS. TRREE is published by Ontotext only under a proprietary license, and is now a component of OWLIM [15]. The version of TRREE used in TripCom offers a limited OWL 1 support, which can, however, be extended by custom rules. [SMS08]

The security concept is based on tuple's labels, which are used for the decisions concerning user permissions.

Data Model

The data model used to store the state of a space into a triplestore is formally defined in a metadata vocabulary—the TripCom ontology [KSM+07].

The only possible data structure for tuples is a triple, which can be tagged by labels (URI or literal). These in turn are used as metadata, used for example for security restrictions. Except for TRREE, the tagging functionality on a triple level—called a tripleset—is not supported by any other semantic framework and is not included in any standard. Because a named graph cannot contain two triples with the same data, two tuples with the same content in the same space are not possible. Kernel components, or in some cases an administrator, can create additional metadata for distribution, replication, security, and logs.

A space is modeled by a named graph, which contains its tuples as triples. Furthermore, a space can have several sub-spaces, which are structured into a hierarchical tree. The data of spaces are not allowed to overlap. For each server domain only one root space can exist. The URI of a space is the concatenation of the host name of the hosting kernel, the name of the parent space, and the space's name.

To read tuples, the user has to define a SPARQL SELECT query, which has to return tuples (triples) as a result.

Architecture

The architecture of TripCom is described in the deliverables [MJK+07], [LSC+09], and [KSdf+08].

The principal idea of TripCom was to create a system that can be seen as a virtual super store. TripCom differs between the client and the server nodes. The server nodes host the space infrastructure and the data. Clients communicate with it by using the TripCom API with any server, and do not have to care about the distribution of the data. Server delegates the write and read operations to the responsible server. All server nodes share together a distributed hash table (peer-to-peer network overlay) for distributing the tuple indexes of all spaces. Client requests with no specific space are routed to the most suitable server based on the information of these tuple indexes.

A server instance—called a kernel—has a modular architecture. It is divided into several components, which are connected over a kernel-internal space, implemented in JavaSpaces

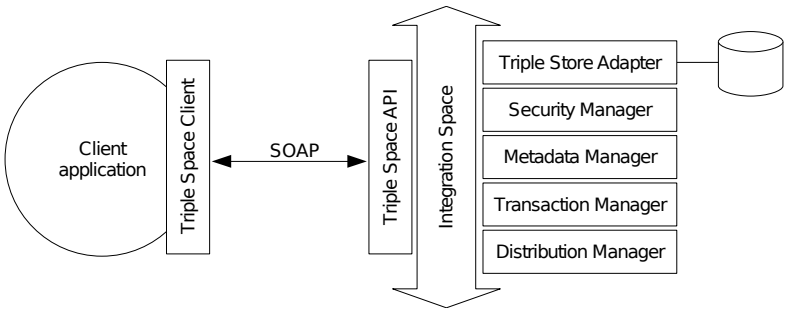

Figure 4.1: TripCom architecture [LSC+09]

[FHA99]. Thus the components of a kernel can be even distributed in a local cluster. TripCom implements all functionalities of a tuple space. It does not extend any existing tuple spaces framework.

The servers communicate with each other over SOAP. Because there is no hierarchy among the servers, the infrastructure has no central, single point of control. Servers can leave and join anytime. Furthermore, a server is addressed through its public domain name with DNS. Figure 4.1 gives a simplified overview of the communication between a client and a server, and shows the most important kernel components. The Triple Store Adapter is responsible for the communication of kernel components and the triplestore, which persists data and processes queries. The Security Manager validates each user request for verifying security policy violations. The Metadata Manager stores the structure of all spaces and attaches metadata to tuples, for example publishers of data, and access logs. The Transaction Manager creates and manages kernel transactions. The Distribution Manager forwards requests to the responsible kernels. The Triple Space API receives all requests for a kernel via the SOAP protocol and calls the required kernel components.

Transaction

TripCom offers a transaction support, but only in the context of a single kernel. In the conceptional phase, distributed transactions were planed, but were not realized. TripCom has no own isolation manager; instead it uses the one of TRREE. Transactions can be created and shared with other users through the TripCom API, and can be used for nearly every API operation. [LSC+09]

Reasoning

The reasoner TRREE updates all inferred data at each write or delete, which results in slow entry updates and fast queries. For reasoning, the full content of the repository has to be loaded into the memory. [SMS08]

A TripCom space does not automatically perform consistency checks. Such checks have to be done by the user. Consistency check is done for read graphs before they are returned to the user as result. [LSC+09] A read operation returns user-inserted tuples (triples) and inferred tuples.

The user cannot distinguish between these two kind of tuples. If the user tries to delete inferred tuples, nothing happens.

Further Functionality

The same template used for read operations can also be used to register a subscription. The user is notified as soon as the predefined SPARQL query of the subscription matches the data repository.

Some draft concepts for mediation were done, but neither a final design nor any implementation are available.

Use Cases

The TripCom project developed two use case applications.

The subject of the first application was Digital Asset Management (DAM). It was mean to show the use of TripCom as an enterprise application integration (EAI). A digital content marketplace was created as a prototype for distributing digital content from different providers to customers possibly using different communication networks.

The prototype for the second use case provided an infrastructure for sharing health data (electronic patient summaries) between different medical institutions in Europe.

4.4 Semantic Tuple Centres

Semantic Tuple Centres [NOV13] [Nar11] has another approach for enriching tuple spaces with Semantic Web technologies. It is an extension of TuCSon (Tuple Centres Spread over the Network) [OZ99], a tuple spaces implementation written in Java. TuCSon implements ReSpect [Omi06] to handle specification tuples, which describe reactions to special events. This concept is similar to XVSM aspects. Furthermore, TuCSoN offers an extended role-based access control.

In Semantic Tuple Centres, a semantic tuple has to be exactly one instance of a user-defined OWL class (called a concept). For each tuple, the user must declare an individual name, which is used to generate the URI for storing a tuple as an RDF resource into a triplestore. A tuple consists of properties with either a literal value, or an individual name to model a relationship with another tuple. Complex properties are not possible, because blank nodes are not supported.

Pellet in combination with Jena is used as the reasoning and querying engines. At tuple insertion, an OWL reasoner checks the consistency of the tuple against the ontology of the space, and, inside Pellet, its inferred data are produced. Inferred data are used to extend the tuple selection capabilities, but they are not readable for the user. At start-up, OWL ontology is loaded from a file. Tuples are selected by a semantic template, which is converted into a SPARQL query. Semantic Tuple Centres does not support transactions.

As an example of its application, a coordination infrastructure for electronic health records (EHR) in the area of e-health is presented. This scenario was chosen to show the distribution, interoperability and security features of Semantic Tuple Centres.

4.5 Comparison

Table 4.1 shows the differences of the presented solutions. Further comparisons of semantic tuple spaces can be found in [NSKMR08].

	sTuples	*TSC*	*TripCom*	*Semantic Tuple Centres*
Tuple model	resource graph	named graph	triple	resource with literal properties
Tuples with same data possible?	yes	yes	no	yes
Selector	own template	N3QL template	SPARQL	own template
Reasoning	yes (DAML+ OIL)	no	OWL 1 partially	yes
Consistency check	yes	no	yes, but for only read	yes
Transaction	yes	yes	yes	no
Subscription	entry level	entry level	space level	entry level by using ReSpecT reaction
Tuple spaces	JavaSpaces	Corso	own implementation (JavaSpaces kernel internal)	TuCSon
Triplestore	-	YARS	TRREE (Sesame)	Jena
Reasoner	RACER	-	TRREE	Pellet
Active time	2004	2005 – 2007	2006 – 2009	from 2010

Table 4.1: Comparison of reviewed semantic tuple spaces

Even though some semantic tuples spaces offer a transaction support, a concrete solution for long-lived, concurrent transactions was not presented in any of the reviewed projects. No solution supports transactions with timeouts, and only TSC offers long-lived transactions.

The data model selected for a tuple is essential for the kind of information that can be stored in a space. There exist three different kinds of semantic tuple structures: (1) a tuple is only an RDF statement, (2) an RDF resource is representing a tuple, and (3), for each tuple, an extra RDF named graph is used. In all the solutions, the tuple ID is publicly accessible, and can be used to reference between tuples. In TSC, the tuple's ID is generated on server-side, which stands in contrast to Semantic Tuple Centres, where the user defines the ID. In the first case, the user does not have to worry about a space's unique ID, but cannot insert an existing RDF graph by splitting it into several tuples.

Two of the solutions are not able to store every kind of data permitted by tuple spaces. TripCom does not allow two tuples with the same content. TripCom has an enormous overhead

for metadata, because they are stored for every user-written triple. Semantic Tuple Centres does not support a nested data structure for a tuple.

The new version (2013) of SPARQL was published after the reviewed projects were finished. This version was extended with new possibilities (e.g. property path). There is no flexible ontology management offered by any of the reviewed projects, neither a scalable reasoning strategy. All data of a space are loaded for reasoning into the memory at once.

The focus of the reviewed projects was on managing a distributed shared knowledge base. In contrast, the focus of my thesis is on a semantic communication middleware.

CHAPTER 5

Design

This chapter presents the design of Semantic XVSM, including the argumentation for all decisions. It is independent of any programming language. Its implementation and the MozartSpaces specifics constitute a part of the next chapter. The recommendations for using Semantic XVSM are described in chapter 7.

The first section lists all the requirements and the assumptions that have to be fulfilled by the design.

5.1 Requirements

This section gives a more detailed description of the design requirements than that offered in the introduction.

- The use of different triplestore implementations, especially Jena and Sesame, should be permitted. Therefore, every interaction with a triplestore should be compatible with the Semantic Web standards (RDF, OWL, and SPARQL).
- Data should be optionally persisted, and to save memory and disk space, redundancy should be avoided.
- The isolation management should deal with long-lived, concurrent transactions. Because SPARQL does not support any transactions, every insert and delete operation should take into account the fact that after a system crash a consistent state with only committed data can be recovered. Recovery of transactions after a system crash or a reboot shall be a part of future work.
- A semantic extension should not restrict XVSM in any way. The coordinator chain has to be fully supported by a semantic coordinator, and a container should be able to consist of common XVSM and semantic entries. Additionally, different entries with the same value in the same container have to be supported. New API operations have to be avoided.

- With regard to the number of entries per container, the whole design should be scalable, especially the reasoning, which is the most expensive task.

- Entries should be enriched by OWL reasoning. Before an entry is inserted into a container, its consistency should be validated against the container ontology. Furthermore, a flexible ontology management should be offered.

- The coordination-based access control model of XVSM allows restrictions on entry level. Therefore, the data of the blocked entries are not allowed to be the decisive factor for the selection of an entry. Otherwise, a user can guess the values of the blocked entries by trial and error. A SPARQL query for entry selection should only be allowed to use the data of the entries that were validated by the access controller and the transaction manager.

- The design should be optimized for applications with a higher frequency of read, rather than write operations, and with a higher probability of commits than of rollbacks. Hence, read operations and commits should be prioritized in performance decisions.

5.2 Semantic Enrichment of XVSM

The simplest way of combining the two technologies would be to use XVSM as a communication middleware between the users and a triplestore. However, this thesis attempts to enrich the coordination capabilities of XVSM; thus it focuses on dynamic data and not on managing global knowledge. Such dynamic data can be messages, states of resources, process data, and events. Therefore, the semantic technologies should be integrated into XVSM to achieve new selection and notification options. In order to be able to use SPARQL queries to select entries from a container, all entries have to be stored in a triplestore.

Applications that need static data, like facts or knowledge, should store them in a separate triplestore. In cases where static data are needed for reasoning, they have to be included into the container's ontologies. More details about reasoning can be found in section 5.8.

The integration of a triplestore into an XVSM core is mainly needed for querying entries with SPARQL. Additionally, because a triplestore regularly supports persistency, it is used as the single persistence back-end for Semantic XVSM. This has several reasons: First, the triplestore has to load all entries, anyway. Secondly, in order to support large containers that do not fit into the memory, a triplestore has to persist them in any case. Lazy loading of entries and weak references, as presented in [Zar12], are used. A second storage would do away with the requirement to avoid redundancy. Thirdly, if the entries are persisted in another form, the recovery at start-up may take a long time. If the inferred data of entries are not persisted, its duration will depend on the number of entries and the complexity of the reasoning that needs to be redone. One drawback of using triplestores with insufficient transaction support (e.g. Jena and Sesame) for persistency is that the design of Semantic XVSM is more complex and the semantic data model has to be extended with transactions. Another drawback is the strict dependency of the semantic coordinator with the semantic persistence back-end. Thus, no other XVSM persistence back-end can be used. But most triplestores themselves offer different persistence strategies. Jena and Sesame have both two

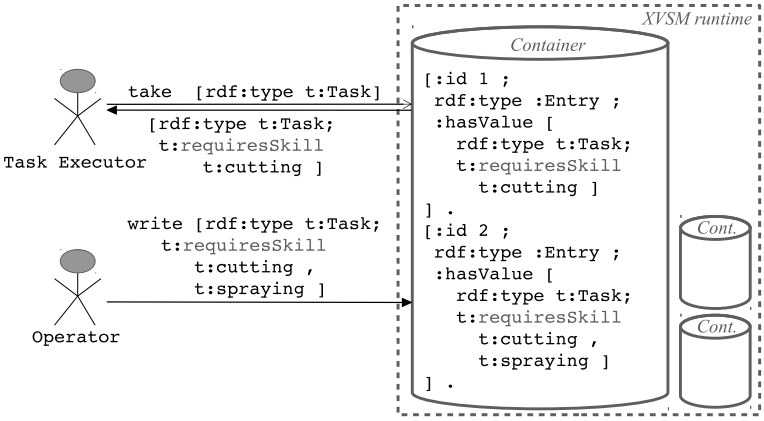

Figure 5.1: Take and write semantic entries

fully supported persistence back-ends: an in-memory, and a native storage. More details about transaction handling are found in section 5.6.

Figure 5.1 shows in a simplified form the interaction with Semantic XVSM. This example presents taking and writing of semantic entries. It is borrowed from the use case presented in chapter 8 about task allocation. Because of limited space, the tasks are simplified and `http://xvsm.org/semantic#` is used as default name space, and `http://xvsm.org/tasks#` is represented by the name space "t". The first user sends a take operation including a template to get entries of the type `t:Task`. A task that requires the "cutting skill" is returned as result. The second user writes into the container a task, which requires the "cutting skill" and the "spraying skill". A cylinder is the graphical representation for an XVSM container. It contains its state, represented by an RDF graph.

5.3 Data Model

The most crucial part of the design is the data model. It defines the manner in which the whole state of an XVSM core is stored (including all entries) in a triplestore. The options for the design decisions given in the following sections are highly influenced by the decisions presented in this section.

First of all, a short summary of XVSM entities and Semantic Web technologies should expose their similarities.

The main entities of XVSM are: a core, a container, and an entry. A core describes a single runtime instance. One core contains several containers, which themselves contain different entries. A container is hosted in its entirety by a single core. An entry can be a part of only one container. It consists of a value and several coordination data. Each coordinator can attach to an entry its coordination data as an X-tree, labeled by the coordinator's name. Committed written entries are not changeable. Different entries with the same value can occur in a single container. All data of an XVSM core, including its entries, are structured in one big X-tree, with the core as root. An X-tree has a recursive data structure, where each node is either a sequence or a multiset (a

set that allows duplicates) of labeled X-trees, or a literal. According to the XVSM specification, each coordinator can also have coordination data on the container level, used mostly for efficient indexing. These data are ignored in the first version of the semantic data model, because usually they can be reconstructed from the entry coordination data. In summary, an entry is an X-tree, of which the sub-nodes are its value and labeled coordination data.

RDF is a framework to describe resources. A resource can be everything, which can be identified by a URI. A resource is described by a set of triples. An anonymous resource—also called a blank node—has only a temporal identifier, which is only unique inside a document or a triplestore. Furthermore, SPARQL introduces named graphs, which are sets of triples identified by URIs. A triplestore hosts several named graphs and offers a SPARQL query engine.

After this short repetition of the background technologies, the data model of Semantic XVSM is presented. The namespace for the ontology entities defined for the Semantic XVSM data model is `http://xvsm.org/semantic#`. In this thesis the prefix label `sxvsm` is used for this namespace.

A tree is only a special type of graph. Hence, an X-tree can be modeled as an RDF graph. The only difference is that the labels of an X-tree have to be mapped to URIs, so they become properties. A nested blank node has always a tree structure. Therefore, it is the preferred structure for modeling of an X-tree. An X-tree exclusively owns all of its sub-nodes. Similarly, a nested blank node has a single reference to all of its sub-nodes. A complete nested blank node can be read out of a triplestore, by following all of its sub-nodes that are blank nodes. An X-tree multiset (also called a bag) is mapped to an RDF bag container [BG04]. An RDF collection is used for a sequence in an X-tree.

Resource Tree

This subsection introduces the definition of the *resource tree* data structure. In this thesis, this structure is used as the basis for additional data structures.

A resource tree is an RDF graph that has a tree structure. All sub-nodes of the root node have to be exclusively owned by the resource tree. In contrast to an XVSM X-tree, a sub-node is not allowed to be a resource tree itself. Each resource tree must have a type, which is defined at the root node level. Before a resource tree is written into a triplestore, the property `sxvsm:id` is attached to its root node.

Entry

Entries are modeled by resource trees. Because each triple has to be assignable to a specific entry, a tree structure is optimal. In this thesis, a resource tree that represents an entry is called an entry resource.

An entry value—either a nested blank node or a literal—is added as a property (`sxvsm:hasValue`) to the entry's resource. Values (e.g. plain Java objects or RDF graphs) that cannot be mapped to a nested blank node are serialized to a string literal. In chapter 6 a mapper for objects of annotated Java classes to nested blank nodes is presented.

As in the XVSM specification, coordination data are the branches of the entry's tree.

Inside an XVSM core, an XVSM entry has an internal, unique ID. This ID is generated by the server process that handles the write operation. If an entry is removed from a container and then written back again, it will become a new entry and will get a new internal ID. This is the method for updating entries. In the case a new entry has the same key as the old one (that is when two entries with the same key exist in the same container) the XVSM isolation manager, based on the transaction of a user request, decides which entry is the correct version. The internal ID of XVSM should also identify entries inside the triplestore. Therefore, the ID is attached to the RDF resource of the entry as a property (`sxvsm:id`). Each entry resource has to be the type of `sxvsm:Entry`. Listing 5.1 shows an example of a semantic entry with the ID 1. Its value has two properties and the key "12" is its coordination data from a Key coordinator.

```
[ sxvsm:id 1 ;
  rdf:type sxvsm:Entry ;
  sxvsm:hasValue [
    t:prop1 "value1" ;
    t:prop2 [ t:prop22 2 ]
  ] ;
  sxvsm:key "12"
]
```

Listing 5.1: Semantic entry

Because the user cannot use an entry's URI as a public identifier, external references have to be done on the coordination data of an entry. For example, with a Key coordinator, a key can be attached to an entry. This key can, of course, be a URI. If the user wants to attach metadata to entries, she/he has to use the coordination data. More details can be found in section 5.7.

It is important to explain shortly, why some other entry models from the past projects were not used in this thesis. Named graphs were not chosen for the entry representation, because they are needed for modeling XVSM containers. Furthermore, named graphs were not used, because a container should be able to store a huge number of entries, and Jena is not optimized to handle a huge number of named graphs. Additionally, pure triples were not selected, because they cannot represent the same information as XVSM entries, cannot have a practical identifier, have excessive overhead for metadata, and do not permit different entries with the same value.

Container

The equivalent of an XVSM container is a named graph. This is because inside its single instance an entire entry resource is distinguishable from other entry resources. Its URI is a combination of the scheme `container` and the container's ID (e.g. `container:1`). Like in the XVSM specification, the ID is for the internal core use only. If a container should be publicly addressable, then its public name should be used. To be able to use DNS to locate a container, its name can be combined with its host name (e.g. `xvsm://host1/tasks`).

In this design, coordination data on the container level are not respected. They would be also modeled as resource trees, and are stored in the named graph of their container. The update and transaction handling must be similar as for entries. In the current version of MozartSpaces, these

coordination data are not needed by any default coordinator, except for the vector coordinator. With a special data structure for the indexing of entries, an implementation of the vector coordinator that uses coordination data only on the entry level should be possible. This structure must have a defined order. It must enable the insertion between arbitrary indexes and the removal of any indexes. Such a structure could be a special tree, where a new index between ”1” and ”2” could be ”1.1”. To create a new index between ”1” and ”1.1” a negative number is needed, like ”1.-1”. Future works should validate this structure and find such a replacement for the vector coordinator.

Core Metadata

The meta model in the XVSM specification is structured as one big X-tree. The root is the universe, with XVSM cores as branches.

The data model used for storing data into a triplestore does not include any data on the universe and the core level. However, all parts of the XVSM meta model that are relevant for the selectors and for restoring are included.

Container descriptors are stored in the named graph with the URI `xvsm:core-data`. Each container owns a named graph that includes all its entries.

The structure of a container descriptor is a resource tree. It contains the same information needed for the creation of its container: the name of the container, its maximal size, and its coordinators. For each coordinator the following must be known: its obligation, its type, and its specific parameters.

Listing 5.2 shows the descriptor for a container with the ID 1. Its name is ”tasks’, it can contain maximal 1000 entries, and has two obligatory and one optional coordinators. In all branches, every property except of `rdf:type` is optional. Some details differ form the XVSM specification, for example `cref` is called `sxvsm:id`.

```
[ sxvsm:id 1 ;
  rdf:type sxvsm:Container ;
  sxvsm:name "tasks" ;
  sxvsm:maxSize 1000 ;
  sxvsm:obligatoryCoordinator
    [ rdf:type sxvsm:key ; sxvsm:name "task_number" ] ,
    [ rdf:type sxvsm:fifo ] ;
  sxvsm:optionalCoordinator
    [ rdf:type sxvsm:label ]
] .
```

Listing 5.2: Container description

Entry Metadata

In XVSM, coordinator-specific metadata can be attached to entries by the user, or the coordinator itself. Because the semantic coordinator is very general, it does not attach implicit data itself,

but it enables the user to bind with the entries labeled X-trees as metadata. One needs to exercise caution to prevent the use of the same label by a user and a coordinator. Because new coordinators can only be created during container creation, only a restriction, or a check for the user-defined labels is necessary. User-definable labels could be restricted to a special namespace (e.g. `http://xvsm.org/custom/`). This solution, however, would forbid the use of existing ontologies. Therefore, in every write operation it is necessary to check if a user-defined, assigned to a coordinator-label is included. The information needed for this check can be found in the container descriptor.

To attach implicit data (like a time-stamp) to an entry, an extra custom coordinator should be used. These data are also accessible from a semantic query. The dependency of such semantic query to another coordinator and its internal data model can cause several problems. Additional safeguards should be a part of future work.

5.4 Read Resource Tree from Triplestore

This section presents a SPARQL query that reads a complete resource tree (entry, or container descriptor) at once. This is necessary, because a nested blank node cannot be read partially. As soon as a reference to a blank node from the triplestore is read, it becomes valid only in the context of the response document.

The simplest algorithm for reading resource trees would be to recursively follow all sub-nodes that are blank nodes. But as shown in chapter 3, the only recursion offered by SPARQL is the property path. However property path decelerations cannot make restrictions on the value of the properties included in the path. This restriction is not necessary anyway, because in a named graph with resource trees, data structures other than resources tree are not allowed, and a resource tree cannot have a direct reference to another resource tree. Therefore, an algorithm can follow all branches without checking for blank nodes.

The query in listing 5.3 reads the entry with ID 1, including its complete value, but without its coordination data (metadata). In contrast, the query in listing 5.4 includes also the coordination data. Because the ID is only for internal use, it is not read. By changing the RDF type to `sxvsm:Container` and the named graph to `xvsm:core-data`, this query can be also used for reading a container descriptor.

5.5 Persistence

In many applications, the data of XVSM containers should be persisted. This way, huge datasets, which do not fit into the memory, are possible. After a restart or a crash such data are still available.

In some cases, persistency is not necessary. For example for the containers used only for caching, or for other temporal tasks. Hence, the persistency should be optional on the container level. If the selected triplestore implementation does not support different persistence profiles for different named graphs inside the same runtime instance, then at least one triplestore instance for in-memory and one for persistence is necessary.

```
CONSTRUCT {
  ?entry sxvsm:hasValue ?v ;
    rdf:type sxvsm:Entry .
  ?s ?p ?o .
}
WHERE {
  GRAPH <container:1> {
    ?entry sxvsm:id 1 ;
      rdf:type sxvsm:Entry ;
      sxvsm:hasValue ?v .
    OPTIONAL {
      ?entry sxvsm:hasValue/(!rdf:nothing)+ ?s .
      ?s ?p ?o .
    }
  }
}
```

Listing 5.3: Read entry without coordination data

```
CONSTRUCT {
  ?entry ?c ?v ;
    rdf:type sxvsm:Entry .
  ?s ?p ?o .
}
WHERE {
  GRAPH <container:1> {
    ?entry  sxvsm:id 1 ;
      rdf:type sxvsm:Entry ;
      ?c ?v .
    OPTIONAL {
      ?entry (!rdf:nothing)+ ?s .
      ?s ?p ?o .
    }
  }
}
```

Listing 5.4: Read entry including its coordination data

In this thesis, some design decisions regarding persistency differ from the decisions presented in the thesis of Zarnikov [Zar12]. First, query capabilities of the database (triplestore) can be used by selectors. Second, all data are explicitly bound to a container, and in case they are related to an entry, they are also bound to its entry. Third, because transaction handling of some triplestores is insufficient, the semantic data model deals also with transactions. The transaction management of a triplestore is used to guarantee the consistency of the semantic data model.

5.6 Transactions

One of the strengths of XVSM is the long-lived, concurrent transaction support. The design has to consider the fact, that on the one hand both Jena and Sesame offer isolation only for single write transactions, and on the other hand the SPARQL standard does not include any transactional support. Therefore, a triplestore transaction should live as shortly as possible, and should be used only for essential tasks.

The Semantic XVSM design of this thesis does not support restoring of transactions after a system crash. The only reason for this decision is to get a simpler design, and because the persistency extension of MozartSpaces [Zar12] does the same. To offer a serious long-lived transaction support, supporting the recovery would be necessary, because system crashes during the life-time of a long-lived transaction are probable. In the case where uncommitted data are stored in the triplestore, to reach a consistent state after a system crash, the recovering mechanism at start-up has to remove all of such uncommitted data. Transactions should be atomic, everything or nothing should be executed. Both isolation levels read committed and repeatable read should be supported. Furthermore, the architecture should be optimized for application scenarios with more commits than rollbacks.

In this design, the transactional isolation is done on the level of resource trees. The following design decisions for entries can be applied also to other resource trees (container descriptors).

The XVSM isolation manager determines that entries are written immediately to the XVSM persistence back-end, while remove operations are only performed at commit. For each entry access, the isolation manager assigns the visibility of the entries to a concrete transaction. An XVSM transaction cannot be mapped to a triplestore transaction, because it would require more than one active write transaction in the triplestore. Creating snapshots of a triplestore for each transaction is not supported, because it would take too much time and memory, especially for huge containers. Therefore, to clean up unfinished transactions after a system crash, only an extension of the semantic data model remains as an option. The drawback of the extended data model is that reasoning over all data of a container at once is practically not possible, because it would have to deal with the issue of transactional isolation.

There exist two different ways for extending the data model with write-locks: either an entry can be labeled with the transaction's ID, or data can be written temporally into an additional named graph (e.g. `container:1/tx1`) and moved to the container's graph at commit. Because at commit moving all data of a transactional graph to the container's graph would take longer than removing only the write-locks of a transaction, the first solution fits the requirements better. Furthermore, in anticipation of the planned transaction recovery after a system crash, delete-locks will become necessary. These delete-locks can be done by a label similar to the write-lock label.

In the current design, delete-locks are not necessary, because they are not needed for cleaning up transactions. The queries for selecting entries can ignore the lock labels, because the XVSM isolation manager can filter out the hidden entries of the query result. An additional reason for ignoring isolation in queries in the triplestore is as follows: if a query wants to know the visibility of each entry, then it has to consider the write and delete locks, as well as the isolation level for the operation. But even then the behavior of the query would not be correct, because no entry locked exceptions would be thrown in the case of a repeatable-read and a delete-lock.

Each sub-transaction should keep all triples it has already inserted to the triplestore to allow their removal in the case of its rollback. In the case of a rollback of an XVSM transaction, all entries with the write-lock label of the transaction are deleted. Remove operations are executed during the commit of an XVSM transaction.

When a container is created, a write-lock label—the same as for entries—is attached to its descriptor in the `xvsm:core-data` graph.

Some SPARQL functions for aggregation or count will not make sense anymore. Because on the triplestore level XVSM transactional isolation is ignored in queries, invisible entries from other transactions may be included in aggregations. To restrict the count of entries in the query result, the count of the XVSM selector should be used. Then, if the triplestore supports streaming for results, the next matched entry will be read until the count is reached. Thus, the query engine does not have to do needless work.

All data of an entry, including its inferred data, must be written at once to the triplestore, because a nested blank node cannot be split and be written in parts. Because reasoning shall include the coordination data of entries, it has to be done after the processing of all coordinators. Therefore, an extra hook in the XVSM write operation is necessary for processing reasoning and for inserting the entry into the triplestore. This hook should be placed after the coordinator processing, and before the check of the access control.

During the commit of an XVSM transaction, all update operations (removing write-lock labels and deleted entries) have to be processed in a single triplestore transaction. Otherwise, a system crash can cause an inconsistent state in the data model, and the transaction would not be atomic. If the processing is done with SPARQL, then all operations have to be included in an update request. During the rollback of an XVSM transaction, the remove of the uncommitted entries can be done in several transactions.

SPARQL does not guarantee any order for queries without an order-by clause. This fact has to be considered when using repeatable-read transactions. To prevent conflicts, an explicit order-by can be of help.

In some cases, a better concurrency can be achieved by using a single triplestore instance for each container. There would be still one XVSM transaction manager in a runtime. This strategy needs an additional mechanism to guarantee a clean commit (and rollback) over several persistent containers. If during the commit a system crash happens, the commit should be finished during the next XVSM core start-up. To that end, additional persistent meta-data would be necessary.

5.7 Querying Entries

One of the deciding reasons for using Semantic XVSM is the selection of entries with SPARQL queries. It results in a new semantic coordinator/selector being created, which can still be combined in a chain with other XVSM selectors. The result of a query is a list of entry's IDs, which then is converted to a list of lazy entries. The value of a lazy entry is loaded from a triplestore only on demand. If a follow-up selector filters entries out, then their values are not loaded needlessly from the triplestore. The result of a SPARQL query is streamed from the triplestore until the user-defined count is reached, or in the case of `count_ALL`, an entry is blocked by the isolation manager.

The basic structure of a SPARQL query for selecting entries is shown in listing 5.5. The template for selecting entries is, like the entry itself, a nested blank node. The dataset of the query is the named graph of the container (`container:1`).

```
SELECT ?entryid
FROM <container:1>
WHERE {
  [ sxvsm:id ?entryid ;
    rdf:type sxvsm:Entry ;
    sxvsm:hasValue [
      :hasName "cutting task" ;
      :requiresSkill :CuttingSkill
    ]
  ]
}
```

Listing 5.5: A SPARQL query for entry selections

The semantic selector imposes several restrictions to a SPARQL query:

1. The result has to be a list of entry IDs.
2. The dataset of the query should only be the named graph of the XVSM container.
3. For security reasons, and because of transactional isolation, some data in a container are not allowed be to used in a query. Without this last restriction, hidden entries of other transactions could influence the result. The security aspect correlates to the coordination-based access control model, which can restrict the access on an entry level. Therefore, the data of the blocked entries are not allowed to be the decisive factor for the selection of an entry. Otherwise a user can guess, by trial and error, the values of the blocked entries. Because neither the transactional isolation nor the access control can be implemented on a query level, the query template should restrict the possibility that an entry is allowed to be the decisive factor for the selection of another entry. A restriction for denying the access of coordination data in a query will constitute a part of future work.

In contrast to the first two restrictions, it is hard to guarantee the implementation of the third one, while still maintaining all available SPARQL capabilities.

Coordinator Chain

This subsection is divided into two parts. First, query restrictions are discussed that deal with handling the preselected entries of the predecessor and keeping their order. Second, the optimal streaming strategy is presented. A coordinator chain with several semantic selectors should be possible.

Let us solve the first problem. The easiest solution would be to ignore the preselected entries in the query, and to make an intersection of the entries selected by the query with the entries selected by the predecessor. But this solution would be in most cases inefficient. A better option is to add the list of preselected entries to the query. This can be done by the SPARQL keyword `VALUES`, which restricts variables to the values from a set. But `VALUES` does not guarantee any order. For that reason, an extra variable is necessary. Listing 5.6 shows a SPARQL query limited to the preselected entries. In the case of a user-defined `ORDER BY`, the variable `?order` will be unnecessary.

```
SELECT ?entryid
WHERE {
  VALUES (?order ?entryid) {
    (1 2)
    (2 1)
  }
  [ sxvsm:id ?entryid ;
    rdf:type sxvsm:Entry ;
    sxvsm:hasValue []
  ]
}
GROUP BY ?entryid ?order
ORDER BY ?order
```

Listing 5.6: Query limited to the preselected entries of the predecessor

The second problem of the coordinator chain is to decide whether to get the entries of the predecessor all at once, or separately, as a stream. Especially in the case of huge containers, the streaming mechanism is much faster. The XVSM selector interface includes two functions: `getAll` to get all entries at once, and `getNext` to get the next entry of the result stream. An activity diagram for the `getAll` implementation of the semantic selector is shown in figure 5.2. Figure 5.3 presents the diagram for the implementation of `getNext`. In the cases where the selector's count is `count_MAX`, the result stream of the triplestore is left open. To close it, the XVSM selector interface has to be extended by a close function. In all other cases, to check the count requirement, the complete result set has to be read from the triplestore first, before any result can be transferred to the next selector. Inside the `getNext` implementation, the query to check the match of a preselected entry with the user-defined semantic query can be done with the ASK SPARQL query form.

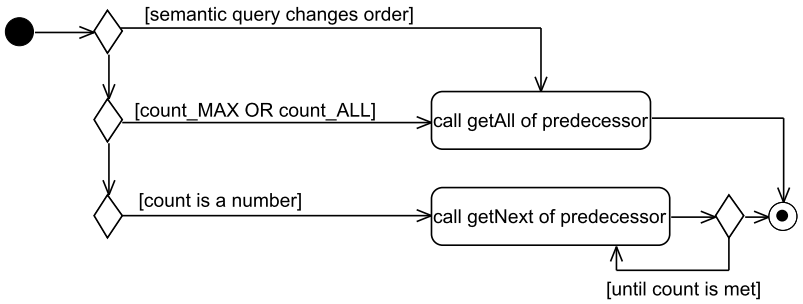

Figure 5.2: Activity diagram for implementing function `getAll` in semantic selector

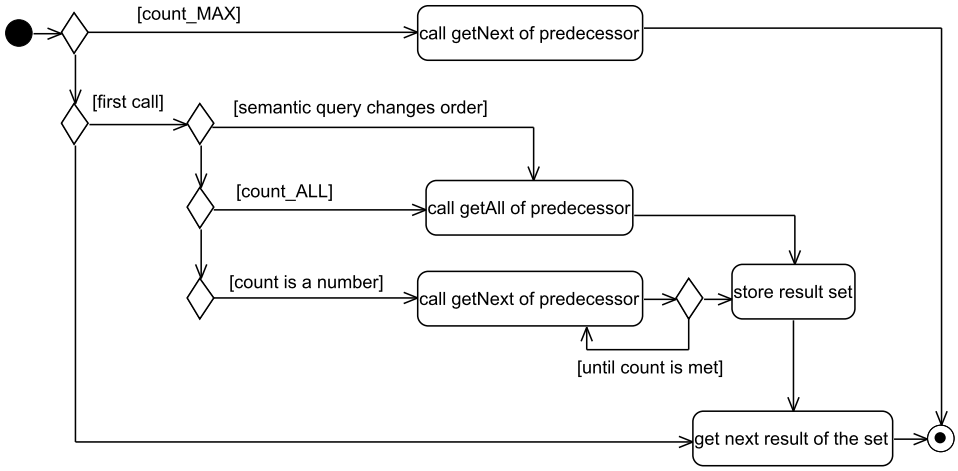

Figure 5.3: Activity diagram for implementing function `getNext` implementation in semantic selector

Query Context

An extension of the semantic selector enables the use of external *context entries* in a query. Context entries can be used as a kind of parameter for SPARQL queries, so that more general and flexible queries are possible. For example, a context entry can describe the state of a software agent. Context entries can be added to a query by the user, or by an XVSM pre-aspect. Such a pre-aspect can, for example, read entries from another container and attach them to a query.

Before a query is executed, context entries are stored in an extra, temporary named graph. A query can use the data of this graph in addition to the data of the container's graph. Listing 5.7 shows an example of such a query. The URI of such a temporary graph uses the schema "temp" and a random number as data (e.g. `temp:2362`). After the query is processed, the temporary graph is removed. During the clean-up after a system crash all temporary graphs should be removed, too. It would, however, be better to use only in-memory graphs. This, however, would require that the triplestore implementation supports in the same runtime in-memory and persistent named graphs. Apart from different variable assignments achieved with the keyword `VALUES`, SPARQL itself does not offer any possibility to attach additional information to a query; only literals and URIs can be stored in such variables.

```
SELECT ?entryid
FROM <container:1>
FROM NAMED <temp:2362>
WHERE {
  [ sxvsm:id ?entryid ;
    rdf:type sxvsm:Entry ;
    sxvsm:hasValue [:name ?name]
  ] .
  GRAPH <temp:2362> {
    [ rdf:type sxvsm:Entry ;
      sxvsm:hasValue [:requiredName ?name] ] .
  }
}
```

Listing 5.7: Query with user context

A query can have several lists of context entries, which are stored in separate, temporary named graphs.

In SPARQL, one uses federated queries to get information from other triplestores. However, the remote triplestore has to offer a SPARQL endpoint via HTTP, and has to be reachable at the time of querying. The triplestores that are used in Semantic XVSM, cannot be used this way, because the isolation and the security are all done on the entry level. Furthermore, not every XVSM core is accessible via HTTP. Moreover, such network accesses can appreciably slow down query execution. In the presented model, the use of SPARQL federation is not restricted and can be used for accessing available endpoints (e.g. DBpedia).

Sub-Query

To use in a query information from other containers (distributed join), sub-queries are introduced. The result of a sub-query is attached to its query as context entries. A sub-query is an XVSM read request that can contain semantic selectors, which can have themselves sub-queries. To distinguish the context entries, each sub-query gets an extra temporary named graph for its result. A sub-query can even access the same container; it helps overcome the third query restriction, that only the selected entry itself is allowed to be the decisive factor for its selection.

In a pre-aspect, sub-queries are processed and their results are attached to the list of context entries of their parent query.

Because a sub-query is a common XVSM read operation, the container location does not matter, and security and isolation aspects are also not a problem. The read access for the sub-queries will use the XVSM transaction of the query operation and not the sub-transaction, because remote access to containers can only by done on the CAPI 4 level, where only transactions are allowed.

Query Structure

This subsection deals with the user-definable aspects of SPARQL query generation. The goal is to limit the SPARQL functionalities only to a degree necessary to fulfill the above mentioned restrictions. Because of the extensiveness of SPARQL queries, the definition of a query construction kit that allows only the creation of legal queries would be too large. It would require an additional user API, which should be avoided. Therefore, the user shall define the data of the `WHERE` clause, and in an XVSM pre-aspect on server-side, the query is validated against the restrictions. The `ORDER BY` and the `HAVING` clause are also user-definable.

The complete query is build by the semantic selector on server-side. The query form SELECT is used. To guarantee the result of an entry's ID list, the query result includes only the variable `?entryId`, and the user-defined `WHERE` part is extended with two triples for defining the variables `?entry` and `?entryId`. The default graph (`FROM`) of the query's dataset is always the named graph of the container. The temporary named graphs with the context entries are added to the dataset by `FROM NAMED` clauses. Additional named graphs are not allowed in the query. For each context entry list, the user has to attach a name for a variable. These variables are bound to the URIs of their temporary graph.

To prevent the use of another entry as the decisive factor for an entry selection, the user-definable `WHERE` part has to be validated against the following guideline: The query template must have a tree structure with `?entry` as the root node. Therefore, all RDF subjects must be either the root node (`?entry`), or one of its sub-nodes. This is the case, if, except for `?entry`, all subjects occur also as an RDF object. The sub-templates of `GRAPH` and `SERVICE` are excluded from this rule. Inside them, all triples are allowed, and their RDF objects are not counted for the tree validation.

A further restriction is that the `VALUES` clause is reserved for the coordinator chain and therefore not available to the user. In the case the semantic selector has a predecessor, a `VALUES` element will be added with all preselected entries.

Furthermore, the `GROUP BY` element is defined by the semantic selector with the variable `?entryid`, and in the case of a predecessor, the selector `?order`.

Because the SPARQL syntax is too extensive, the checker cannot be done by regular expressions. The best way for an implementation would be to use a SPARQL parser, like ARQ of the Jena project.

Because at the query level transactional isolation and access control is ignored, the query selects also entries that are filtered out afterward. For this reason the user has to be careful with aggregation functions and be aware that the LIMIT clause is not supported.

Listing 5.8 presents an example for a more complex query that uses context entries. The user-defined part of this query starts and ends with a code comment. This query can be used for the entries in figure 5.1. The properties `t:requiresSkill` of an entry are counted. Then the overlapping skills from the entry and the context entry are counted. If both count are the same, then the entry is selected.

```
SELECT ?entryId
FROM <container:1>
FROM NAMED <temp:132>
WHERE {
  BIND (<temp:132> AS ?context)
  ?entry sxvsm:id ?entryId ;
    rdf:type sxvsm:Entry .

  # begin of user-definable part
  ?entry sxvsm:hasValue ?entryValue .

  OPTIONAL {
    SELECT ?entryValue (count(?ns) as ?requiredSkills) {
      ?entryValue t:requiresSkill ?ns .
    }
    GROUP BY ?entryValue
  }
  {
    SELECT ?entryValue (count(?ts) as ?overlappingSkills) {
      ?entryValue t:requiresSkill ?ts .
      GRAPH ?context { ?contextEntry t:hasSkill ?ts . }
    } GROUP BY ?entryValue
  }
  FILTER (!bound(?requiredSkills)
    || ?requiredSkills = ?overlappingSkills)
  # end of user-definable part
}
GROUP BY ?entryid
```

Listing 5.8: An example for a more complex query using context data

To summarize, a semantic query (semantic selector data), which the user defines, consists of the user-definable `WHERE` clause part, the `ORDER BY` part, the `HAVING` part, lists of context entries, a list of sub-queries, and the selector's count.

Stored Query

To enable the management of the whole coordination logic in the semantic space, semantic queries can be stored as entries into a container. To execute such a query, the user sends a list of XVSM selector data for getting the *query entry*, and the query-required context entries as parameters. An XVSM pre-aspect listens for semantic selector data that requests a query entry. Such an aspect

gets the query entry by executing a read operation with the user-passed selector data, and attaches the context entries to the read semantic query. This pre-aspect has to be executed before the one for processing sub-queries.

For identifying query entries, an extra Key coordinator might be useful.

5.8 Ontologies and Reasoning

Beside queries, reasoning is the second key feature of Semantic XVSM. The entries get enriched with implicit data and additional information for coordination, which are retrieved by reasoning. The inferred data can be used by queries, and can be optionally returned to the user. Furthermore, entries, which are classified as inconsistent during the reasoning, are blocked from getting inserted into the container. Each container has its own ontology for the reasoning of its entries. This ontology can be managed at run time.

Ontologies can be used for different things; they can describe the domain model to retrieve implicit knowledge. They can also include coordination logic: For example, entries with specific properties become the type of specific classes, which then can be used in queries for selection. Additionally, ontologies can be used to deny the insertion of inconsistent entries. Beside ensuring the conformance of entries to a data model, this can be used to forbid entries with a specific pattern.

Reasoning, or consistency checks for a whole container offered by the previous projects, are not supported by Semantic XVSM. To support them, either an ontology would have to deal with transactional isolation and access control, which is practically not feasible, or for each transaction an extra snapshot of the triplestore would be necessary, which would require to much memory, and would cost too much time for its creation. Instead, in Semantic XVSM, reasoning is done for each entry separately, before its insertion into the triplestore. This solution has several advantages:

- For a consistency check, reasoning has to be done at the insertion time, anyway.
- Most reasoners, including Jena and Pellet, have to load the complete dataset into the memory, which is not possible for all entries of huge containers.
- A triplestore, like Sesame, without included reasoners, can be used as persistence and SPARQL back-end, because reasoning is done by an extra component.
- For big containers, the reasoning over only a small dataset (a single entry) will scale better than the reasoning over all the data of the container.
- Because reasoning is done at entry insertion, it is possible to make reasoning only for special kinds of entries.

The solution that implies separate entry reasoning has also drawbacks. Updates to the ontology have no impact for already existing entries. To overcome this problem, the reasoning can be redone for existing entries after each ontology update. Because such a process can be really expensive, it should be restricted to only those entries that are potentially affected by the change. Therefore, an XVSM selector could be attached to an ontology entry. A redo process of

reasoning should be a part of future works. A further drawback to separate entry reasoning is that not all possible inferred data can be used, because the data of an entry, including its inferred data, must be still a tree, as other data are not allowed in the container's graph. Otherwise, the data of entries would not be assignable to one entry in the triplestore, anymore.

Reasoning is done for the complete entry, including its coordination data. Ontology designers should be aware that some coordinators can get confused if their coordination data are extended with the inferred data. Therefore, in future works, a mechanism should be implemented to prevent storing ontologies that could cause conflicts with coordination data. Some coordinators, for example a type coordinator, can use the inferred data, such as a class hierarchy. As already written in the section about transactions, reasoning should be done inside a hook in the write operation, after all coordinators have attached their data. In the case of an inconsistent entry, an exception will be thrown, which will abort the sub-transaction.

Each container has its own ontology for its entries. Since an ontology can include facts described by RDF graphs, which could also be of interest to entry queries, the RDF graph of the complete container ontology should be accessible for queries. Therefore, the ontology is stored in an extra named graph (`container:1/ontology` for the container with ID 1)—the *ontology graph*—, which then can be used in a query inside an extra `GRAPH` template. The URI of the named graph of the ontology is bound to the variable `?ontology`. Furthermore, such an ontology graph should also include the data, which can be inferred by reasoning only over the container ontology without entries. These data can also be of interest for querying.

Generating inferred Data

To produce inferred data for an entry, the complete container's ontology and the entry resource (resource tree) is loaded into a reasoner. After processing, the inferred data are retrieved from the reasoner by reading all triples, where the subject is one node in the entry resource. The delta with the original entry data are then the inferred data. If new nodes are generated by reasoning, then they will be ignored. Only the nodes of an entry that already exists in the raw data are enriched with new properties and values. As optimization, the ontology should only be loaded once, and afterward, if the used reasoner supports it, should be used for reasoning of all entries.

In most cases, data will be exchanged with the reasoner in Turtle format, as strings. Some reasoners do not support blank nodes as individuals. In such a case, all blank nodes of the entry resource have to be changed to other temporary URIs before reasoning, and after reasoning they have to be mapped back.

Storing inferred Data

Because the inferred data are produced at the entry write, they have to be stored into a triplestore. But in some cases the raw data are still necessary, so it should be possible to distinguish between the raw and the inferred data. There are two options to do that: First, the entry is stored twice: once with the inferred data, and once without. This solution, however, has the drawback of redundancy. In the second option, the inferred data of all entries of a container are stored in an extra named graph (e.g. `container:1/inferred`). Thus, the inferred data can optionally be used in a query for selecting entries by attaching the inferred graph to the query's dataset by an

additional `FROM` clause. Furthermore, the inferred data will be only included with the returned data, if the user wishes that explicitly. The problem of this second solution lies in the fact, that it may not work in all triplestores, because the option to split a blank node over different named graphs is not defined in the SPARQL standard. But tests have shown that at least in Jena and Sesame this solution works. Therefore, the second solution was chosen for storing inferred data.

The queries for reading and deleting entries that are defined in a previous section, have to be extended for inferred data. Listing 5.9 shows a SPARQL query to retrieve the data of an entry, including the inferred data. A query for deleting is very similar, and is presented in listing 5.10. Because the references of blank nodes are only valid inside one SPARQL update query, entries have to be inserted into the triplestore at once, including their inferred data. The query in listing 5.11 fulfills that task.

```
CONSTRUCT {
  ?entry rdf:type sxvsm:Entry ;
         ?c ?v ;
         ?c2 ?v2 .
  ?s ?p ?o ;
     ?p2 ?o2 .
}
WHERE {
  GRAPH <container:1> {
    ?entry sxvsm:id 1 ;
           rdf:type sxvsm:Entry ;
           ?c ?v .
    OPTIONAL {
      ?entry (!rdf:nothing)+ ?s .
      ?s ?p ?o .
      OPTIONAL {
        GRAPH <container:1/inferred> {
          ?s ?p2 ?o2 .
        }
      }
    }
  }
  OPTIONAL {
    GRAPH <container:1/inferred> {
      ?entry ?c2 ?v2 .
    }
  }
}
```

Listing 5.9: Read entry including inferred data

```
DELETE {
  GRAPH <container:1> {
    ?entry sxvsm:id 11 ;
           rdf:type sxvsm:Entry ;
           ?c ?v .
    ?s ?p ?o .
  }
  GRAPH <container:1/inferred> {
    ?entry ?c2 ?v2 .
    ?s ?p2 ?o2 .
  }
}
WHERE {
  GRAPH <container:1> {
    ?entry sxvsm:id 11 ;
           rdf:type sxvsm:Entry ;
           ?c ?v .
    OPTIONAL {
      ?entry (!rdf:nothing)+ ?s .
      ?s ?p ?o .
      OPTIONAL {
        GRAPH <container:1/inferred> {
          ?s ?p2 ?o2 .
        }
      }
    }
  }
  OPTIONAL {
    GRAPH <container:1/inferred> {
      ?entry ?c2 ?v2 .
    }
  }
}
```

Listing 5.10: Delete entry including inferred data

```
INSERT DATA {
  GRAPH <container:1> {
    _:b1 sxvsm:id 12 .
    _:b1 rdf:type sxvsm:Entry .
    _:b1 sxvsm:hasValue _:b2
    _:b2 example:hasSkill example:buringSkill .
  }
  GRAPH <container:1/inferred> {
    _:b2 example:isAbleTo example:burn .
  }
}
```

Listing 5.11: Insert an entry with inferred data

Ontology Management

To offer an ontology management without extending the XVSM API, an *ontology entry*, a special type of entries is introduced. Each ontology entry holds a piece of the whole container ontology. The union of all ontology entries in a container makes up its ontology, which is used for reasoning of newly inserted entries. Hence, every user can easily read, extend and reduce the ontology. Therefore, new data models can be stored before an entry that belongs to it. Moreover, coordination ontology can be changed at run time.

For simplicity, only committed ontology entries are used for reasoning. Thus, at any given time, for all transactions in a container there exists only one ontology for reasoning. Therefore, ontology management must be done in a separate transaction, before it can be used for semantic entries.

As already mentioned, changes in the ontology have no impact on the existing entries.

The data of an ontology entry are stored as a string literal. An ontology entry has to be a type of `sxvsm:OntologyEntry`. Before an ontology entry is committed, it is treated like a normal entry. But in an XVSM pre-commit-aspect, its data are loaded into the reasoner together with the data of all other ontology entries and are stored with their inferred data in the ontology graph (e.g. `container:1/ontology`) of the container. In the case an ontology entry is removed, the reasoning has to be redone, too, and the ontology graph has to be updated afterward. During the reasoning of an updated ontology, its consistency is checked, and in the case of inconsistency, the transaction is aborted.

For future work, a dependency management designed to prevent removing ontology entries needed by other (normal, query, ontology, or subscription) entries would be desirable.

5.9 Entry Notification

In XVSM, post-aspects can be used to notify about executed operations on entries. The feature presented in this section enables the user to restrict the notification to the entries with user-specified characteristics. In contrast to the thesis of Murth [Mur10], this feature acts on the entry level, and does not issue a notification when a special container state is reached. Such

subscriptions have to be handled by future work. Perhaps the best way to implement container or space state notifications would be to replicate all potential relevant data into an extra triplestore, which state would be observed. For keeping replications up-to-date, the entry notification feature from this section can be used.

Subscriptions are designed to always observe a concrete container. To enable easy management, they are written as special entries (*subscription entries*) into the observing container. Thus, the user does not need any rights to create new aspects, and no additional API extension is necessary. Every time the post-aspect of an entry operation is executed, a semantic read operation looks for the subscription entries that conform with the action and the processed entry. Therefore, a subscription entry must contain the entry type (OWL class), action types (read, take, or write) and a subscription ID. The entry type and actions types are the preconditions that a notification for a subscription is sent. The entry of the processed operation have to be an instance of the entry type. The criteria that this entry has to fulfill are declared in the container ontology. This has the advantage that these criteria can be also used by other subscriptions and queries. The calculation to retrieve the classes for an entry is done only once, at entry insertion, as described in section 5.8. The drawback of the subscription based on OWL classes is that the necessary ontology classes have to be written before the observed entries are inserted.

Listing 5.12 shows an example of a generated semantic query for reading subscription entries. The operation and the list in the filter varies depending on the current operation. In this example, subscriptions are searched for a read operation with an entry that is the type of `http://example.org/TestClass`. The prefix "subs" abbreviates the namespace `http://xvsm.org/semantic/subscription#`.

```
?entry sxvsm:hasValue [
  rdf:type sxvsm:Subscription ;
  subs:conditionClass ?class ;
  subs:operation sxvsm:read
] .
FILTER(?class IN (<http://example.org/TestClass>,
                  <http://www.w3.org/2002/07/owl#Thing>))
```

Listing 5.12: Semantic query for finding subscription entries

Notifications are sent as soon as the notified operation is committed. Therefore, they are written as entries in a notification container, with the same transaction as the notified operation. In the case of an abort of the transaction, nothing will be notified. As soon as the notified action is committed, the notification entry will be visible, and the subscriber will be able to take out the notification entry. Therefore, the subscriber should make a blocking take request on the notification container. Furthermore, as long as no distributed transactions are used, the notification container has to be located on the same core as the observed container. To overcome this restriction, and to support notification containers also on other XVSM cores, the notifications should be written in a local container first. A background job listens to this container and routes the notification entries to the subscriber's container.

The notification message (entry) includes: the type of action, the processed entry, and the

subscription ID. Because the check whether a notification is required is called very often (for every action, in particular), a notification service can easily slow down the space. But the presented solution needs only one average SPARQL query per action, thus the performance loss should be justifiable.

5.10 Summary

In Semantic XVSM, entries are handled as RDF nested blank nodes, which is a representation compatible with the XVSM specification. The entries can be enriched with inferred data, which are produced by reasoning over container-specific ontologies. For entry selections, SPARQL queries with nearly no restriction can be used. Such queries can consist of several sub-queries to support distributed joins.

Special types of entries are introduced: *query entries* for stored queries, *ontology entries* for composing containers' ontologies, and *subscription entries* for notifying about special entry actions.

Figure 5.4 shows a simplified overview of the core components in Semantic XVSM on server-side. The gray-marked component (XVSM CAPI) is part of the standard XVSM.

- *query pre-processor*: An XVSM pre-read- and pre-take aspect is responsible for getting stored queries, for validating queries against the restriction from section 5.7, and for attaching the result of sub-queries to their semantic query.
- *notification emitter*: For each user operation, an XVSM post-aspect looks for subscriptions. For all found subscriptions, a notification is sent.
- *XVSM CAPI*: This component represents the complete XVSM defined by Craß [Cra10].
- *semantic coordinator*: This custom coordinator selects entries by SPARQL. It calls the semantic back-end to execute semantic queries and lets the isolation manager of the XVSM CAPI check the result.
- *semantic back-end*: This component is the adapter between XVSM and the triplestore and the reasoner component. It acts as an abstraction layer and manages persisting of entries and container descriptors, reasoning, and querying.
- *triplestore*: Read and update operations of resource trees, and queries are executed in a concrete triplestore implementation (e.g. Jena).
- *reasoner*: The reasoner component manages a container ontology and produces inferred data for resource trees (entries).

The entire state of an XVSM instance is modeled and stored in a triplestore. The access control concept of XVSM is fully respected. Only new special entry types are introduced, but the XVSM API is not changed for the user.

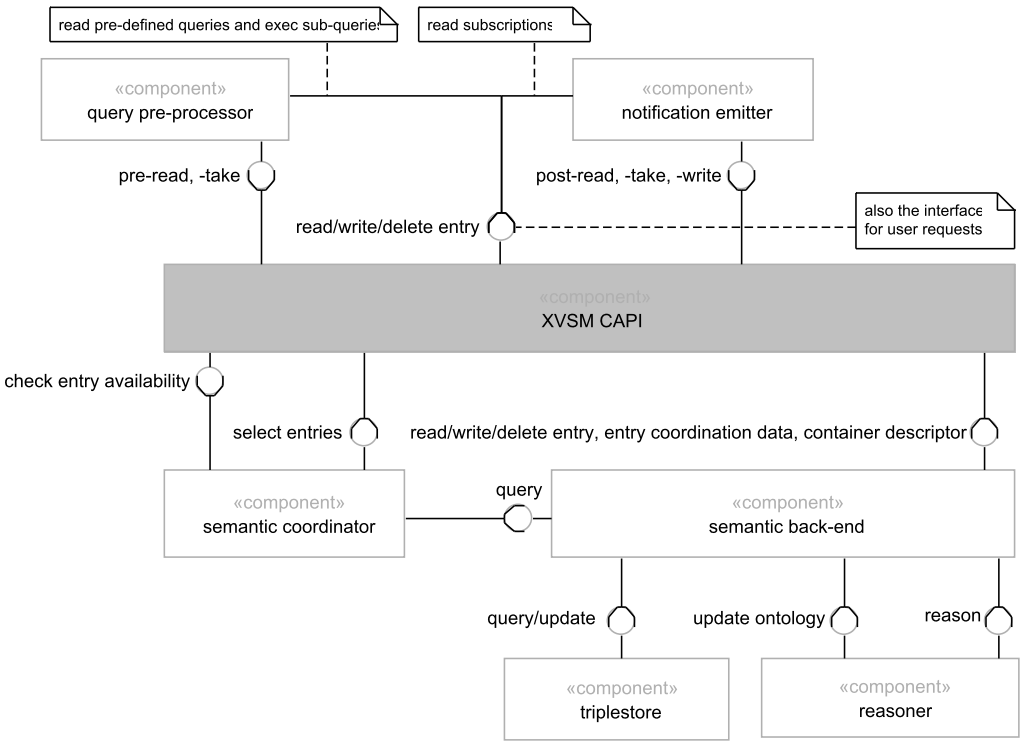

Figure 5.4: Simplified component overview (server-side)

CHAPTER 6

Implementation

This chapter describes Semantic MozartSpaces. It is the reference implementation and the first evaluation of the Semantic XVSM design, which was presented in the last chapter. The implementation extends and adopts native MozartSpaces (version 2.3)—the reference implementation of XVSM written in Java. It is published under the license AGPL [3].

6.1 Architecture

Semantic MozartSpaces consists of four extensions of the native MozartSpaces: the semantic back-end, the semantic coordinator, the query preprocessor aspect, and the semantic notification emitter. Figure 6.1 shows an overview of the architecture of Semantic MozartSpaces. The light boxes belong to the native MozartSpaces or to Jena. The dashed boxes are interfaces with different implementations. The dark boxes are a part of the implementation presented in this thesis.

MozartSpaces *CAPI-3* is responsible for the coordination (coordinators and selectors), transactional isolation, and the management of containers and entries. It combines CAPI-1, CAPI-2, and CAPI-3 of XVSM. *CAPI-4* includes the operation scheduler (event processing), the aspect management, the user API, and the remote communication.

In this thesis, the persistence layer of MozartSpaces described by Zarnikov [Zar12] was adopted, and is now called the *Store API*. It is used by CAPI-3 for storing container descriptors, entries and coordination data. Beside the semantic back-end, there are also in-memory and Berkly DB implementations of the Store API.

The *semantic back-end* uses the Reasoner API for producing inferred data from the entries, and the *Semantic Store API* for storing and querying resource trees. The user can choose between three implementations of the reasoner API: none (a dummy implementation that does nothing), Jena (uses the OWL-1 reasoner included in Jena), and the OWL API. The OWL API has connectors to many reasoners. In the test cases of this thesis, the OWL API is used with Hermit. In the current implementation, a system crash during a commit can cause an inconsistent state in the persisted data. Future works should improve the implementations to guarantee atomic transactions.

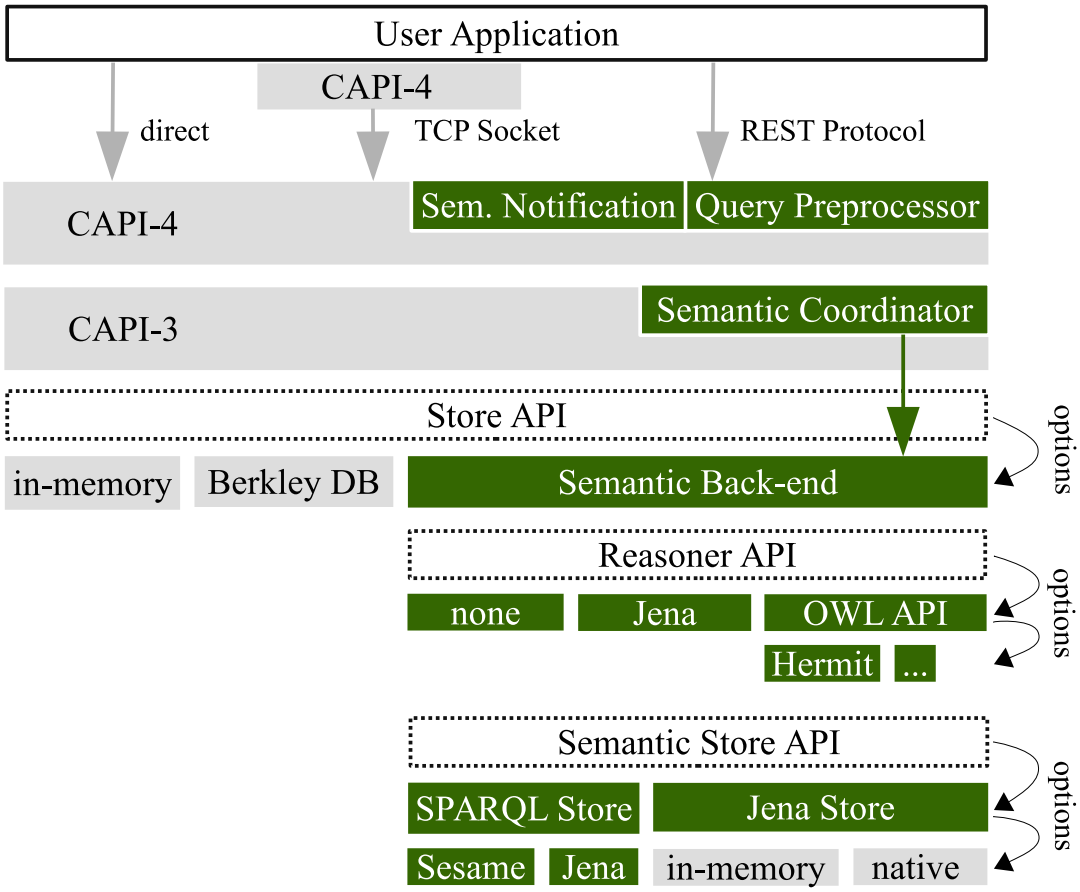

Figure 6.1: Architecture Overview

SPARQL store is an implementation of the *Semantic Store API*. It executes the queries that are described in the last chapter on a *SPARQL endpoint*, which, in turn, is an interface to a triplestore. The *Jena store* implements the same functionality. It uses Jena's native API, and achieves a better performance. Optionally, SPARQL endpoints and Jena store can persist their data. At the moment, in one Semantic MozartSpaces runtime, only one triplestore instance is used. Thus, only one persistence strategy is possible.

The *semantic coordinator* is a custom MozartSpaces coordinator for selecting entries by using SPARQL queries. It works only in combination with the semantic back-end. The *query preprocessor* is a space aspect for executing sub-queries and loading stored queries. The semantic back-end should only be used in combination with the query preprocessor. The *semantic notification* emitter is implemented by a container aspect. It notifies about operations on entries that match the filter of semantic subscriptions.

A *user application* can access a local MozartSpaces core by calling the CAPI-4 directly. For remote access, either the client uses its local CAPI-4 to communicate over the TCP socket protocol, or the client uses the REST protocol. On client-side, no semantic extension of MozartSpaces is

necessary, but it is recommended for more comfort and for error prevention. One useful tool, for example, is the object resource mapper for mapping Java objects to resource trees.

6.2 Store API

The Store API is an adaption of the persistence layer introduced by Zarnikov [Zar12]. This adoption was necessary for implementing the semantic back-end. The Store API is not focused on semantic data. Implementations of it that use other databases are also possible. The only data structure that the old persistence layer offers is a stored map (`StoredMap`). Its instances belong only optionally and indirectly to a container. In the Store API all stored entities (entries and coordination data) must belong to an explicit container.

The Store API offers the following four data structures. (All of them are similar to a Java map.)

- By using the interface `StoredEntryCoordinationDataMap`, a coordinator can attach to the entries coordination data labeled with the coordinator's name.
- The interface `StoredEntryMap` is only used with the class `DefaultContainer`. It stores all entries of the container.
- The interface `StoredContainerMap` is only used with the class `DefaultContainerManager`. It stores all container descriptors (`PersistentContainerDescriptor`) of a MozartSpaces runtime.
- The structure for coordination data on the container level has to be defined in future works. One of the problems that needs to be resolved would be the transactional isolation.

In contrast to the old persistence layer, data structures of the Store API support write-operations, but only with transactions. All data structures are only used by CAPI-3, where every operation runs inside a sub-transaction. The read operations still do not have a parameter with the MozartSpaces transaction. The results of read operations have to be verified by the isolation manager.

For using old persistence back-ends, a default implementation of every new data structure maps all functions to `StoredMap`. Because stored maps are container independent, the names of the stored maps are extended with the IDs of their containers.

For an interim backward comparability, the interface `PersistenceBackend` is not changed. It still allows the use of the existing persistence back-ends. Implementations of the Store API have to utilize the interface `StoreAPI`, which extends the interface `PersistenceBackend`. In future works, `StoreAPI` should completely replace `PersistenceBackend` and should only take over the functions `init`, `makeEntryLazy`, and `close`.

The use of `StoredMap` is not possible in combination with the semantic back-end. Therefore, it was replaced in all coordinators of the native MozartSpaces with the interface `StoredEntryCoordinationDataMap`.

In the semantic back-end, the interface `OrderedLongSet` is not supported. It does not fit into the data model of XVSM, and it is not compatible with the transactional locking strategy of

XVSM. In the current version of native MozartSpaces it is only used by the vector coordinator for its coordination data on the container level. As described in the chapter 5, special data structure for the indexing of entries, should allow an implementation of the vector coordinator that uses only the interface `StoredEntryCoordinationDataMap`. Future works should find a replacement. Until then, the vector coordinator is not usable in Semantic MozartSpaces.

6.3 Data Model

This section describes all data models of Semantic MozartSpaces that are relevant for user applications.

Resource Tree

A resource tree is an RDF graph with a tree structure. It is implemented by the class `ResourceTree`, and is represented by entries and container descriptors.

For modeling a branch of a resource tree, the class `SubResourceTree` is introduced. It is needed to obtain the value and the coordination data of an entry. In contrast to a resource tree, a sub resource tree is allowed to consist of a literal only.

The class `ResourceTree` uses non-serializable models of Jena's RDF API. The class `SerializedResourceTree` is a textual representation of a `ResourceTree`, and is used for network communications. A resource tree is serialized to the Turtle syntax. Because Turtle has a broad tool support, the client-side does not depend on any specific library. No extra serialization is necessary for using the REST protocol. To easily find the root of a resource tree after deserialization, during serialization the root node should be typed with `sxvsm:ResourceTreeRoot`. The class `SerializedSubResourceTree` is a serialized form or a sub resource tree with branches. Literals do not need an extra serialization strategy. Their XML Schema representation is used.

For serializing resource trees, the class `ResourceSerializer` should be used.

In Semantic MozartSpaces, transactional locking can only be done on the resource tree level. A sub resource tree cannot have a separate lock.

Entry

The MozartSpaces API for dealing with entries was not changed. As in the native MozartSpaces, the class `org.mozartspaces.core.Entry` is used for writing entries. It includes the value and the coordination data of an entry. The result of read operations is only the value of an entry.

Every serializable object can be used as an entry's value. To prevent side effects, such objects should be immutable. Such values are still supported by Semantic MozartSpaces. To be able to use the value of an entry in a semantic query, the value has to be an instance of the class `SerializedSubResourceTree`, or an instance of a class prepared for the object resource mapper. On server-side such Java objects are tried to be mapped to sub resource trees. There are two drawbacks to the server-side serialization: the container-hosting MozartSpaces runtime must know the Java class of the object, and, depending on the network protocol used, serialization problems can occur.

Optionally, read operations can return entry values, including their inferred data. This behavior can be declared in the semantic configuration properties. In future works, the class `ResourceTree` could be extended to separately include raw data and inferred data. Then the user always has the choice to use only raw data, or data with inference data.

Serializable Java objects that are the common entry values of MozartSpaces continue to be supported. These values are converted to RDF literals. A Java object with no equivalent RDF literal is serialized to a byte array, as the default persistence back-end of MozartSpaces does with all objects. This byte array is then converted to a hexadecimal literal.

Caution is necessary, because internally Jena does not differentiate between several number types. This can result in the read value of an entry to belong to a different Java class then the written value. For example, if the entry value has the type `Long` and the value "1", then the result of a read operation is "1" with the type `Integer`.

The semantic back-end stores complete entries as a resource tree in the triplestore. These entries include their values and their coordination data.

Entry's Metadata

The semantic coordinator offers the addition of custom metadata to an entry (even to a non-semantic entry). These metadata can be used in semantic queries. They are attached like coordination data to the resource tree of an entry. The class `EntryMetadata` stores such metadata with their labels. For using metadata, an instance of this class has to be added to an entry as its coordination data.

Semantic Coordinator

For using semantic queries, an instance of the class `SemanticCoordinator` must be registered at the container as a coordinator, either optional, or obligatory. An optional coordinator has less overhead. In a single container, only one semantic coordinator can exist. More than one instance would not make any sense, because a semantic coordinator is stateless.

Semantic Selector and Query

A semantic selector (an instance of the class `SemanticSelector`) includes a count, a semantic query, and lists of context entries. Instead of a query, a list of selectors to read a stored query from the container can also be declared. The semantic selector can also be used to select non-semantic entries by querying for their coordination data.

An instance of the class `EntryQuery` represents a semantic query. By using its data, a SPARQL query is build to select a list of entry IDs on server-side. It includes the user-definable parts of a SPARQL query that were defined in the section 5.7: the user-definable part of the WHERE clause, order-by variables, the `HAVING` clause, and sub-queries.

A semantic selector can include several lists of context entries. Before query execution, each list is stored in a separate temporary named graph in the triplestore. The name of such a graph is binded to a SPARQL variable. This variable name is defined by the user. At present, a context entry has to be of the type of `SerializedSubResourceTree`—the common type

of semantic entry's values. This value is added as a property (`sxvsm:hasValue`) to a new resource tree with the type `sxvsm:Entry`.

A sub-query has to be an instance of the class `SubEntryQuery`. It is a wrapper for a read request that as a result produces a list of sub resource trees. This read request can, of course, also include a semantic selector. The result of a sub-query is attached to the main query as an extra list of context entries.

A stored query is a stored entry with an instances of `EntryQuery` as its value. Context entries that are needed by such a stored query have to be handed over to the semantic selector that should use it.

Ontology Entry Value

The ontology of a container is the combination of all its ontology entries. An ontology entry must have as its value an instance of the class `OntologyEntryValue`. This class is only a wrapper for a `String`, including an RDF graph in a Turtle representation.

Semantic Subscription

Semantic subscriptions are registered by writing special entries. The value of such an entry must be an instance of the class `SemanticSubscription` optionally serialized to an instance of `SerializedSubResourceTree`. It includes a condition class, a subscription ID, notifiable entry operations (read, take, or write), and a reference to the container for writing the notifications. A condition class is an OWL or an RDFS class. To notify about an operation, the value of its entry has to be the type of the condition class.

A notification is also an entry. Its value is a serialized sub resource tree of the Java class (`SemanticNotification`). This class contains:

- the ID of the subscription
- the value of the processed entry
- the processed operation
- the time-stamp, when the notification was created.

The subscription ID is also attached as label to a notification entry. The name of the used label coordinator is "semanticSubscriptionID-label" (stored in the constant `SemanticNotification.SUBSCRIPTION_ID_LABEL`).

Container Descriptor

Container descriptors are used internally in the semantic back-end on server-side. They include all information for restoring a container after a server restart.

The persistent container descriptor (`PersistentContainerDescriptor`) was introduced with the persistence layer [Zar12]. In Semantic MozartSpaces, an object resource mapper compatible class was created. It acts as a wrapper for the old container descriptor. It includes following properties:

- ID of the container
- the maximal size of entries
- classes of obligatory and optional coordinator implementations
- coordinator arguments (Java serialized)
- coordinator restore tasks (Java serialized)
- the authorization level

In future works, a container descriptor should be a pure resource tree without any serialized Java objects. Therefore, the handling of restoring containers has to be changed in MozartSpaces.

6.4 Object Resource Mapper

Semantic MozartSpaces includes a tool for mapping Java objects to resource trees (or sub resource trees) and visa versa. This tool is implemented by the class `ObjectResourceMapper`.

The classes of these objects have to be annotated with `ResourceClass`. They must have a default constructor and must implement the interface `Serializable`. In this thesis, such classes are called resource classes.

The class fields that should be included in the mapping need the annotation `Resource-Property`. These properties must have a public getter and setter.

A registry with Java classes and their corresponding OWL class is integrated in the mapper. A Java class has to be registered with the function `registerClass` before a resource tree (or sub resource trees) can be mapped to an instance of this class. If the root node of a resource tree is of the type of several registered Java classes, the first class will be used. The functions `mapToResourceTree` and `mapToSubResourceTree` automatically register the Java class of the used object.

A resource tree (or sub resource trees) can include more triples than is required to create the corresponding Java object. The additional information will be ignored. If a resource tree (or sub resource trees) includes its inferred data, the mapper will not be confused.

In a resource class a default namespace has to be defined. Optionally, namespaces associated with a prefix can be also defined. Different ways exist to define the URI of resource classes and their properties:

- With the parameter `uri` the absolute URI can be defined.
- With the parameter `name` the relative path can be defined. It is appended either to an explicit namespace or to the default namespace of the resource class. To use an explicit namespace, the parameter `namespace` has to include a prefix defined in the resource class.
- If both `uri` and `name` are empty, the name of the Java field will be used as name, and will be appended to an explicit or the default namespace.

The following Java types are mapped to their equivalent RDF properties:

- A Java *literal* is mapped to its corresponding RDF literal.
- For each entry of a Java *set*, a triple with the following pattern is created: <resource> <property> <set entry>. In the turtle specification it is called an object list.
- Depending on the parameter `listType` declared in the annotation `ResourceProperty`, a Java *list* is mapped to the following RDF types:
  - *collection*: rdf:Collection (default)
  - *bag*: container rdf:Bag (unordered list)
  - *seq*: container rdf:Seq (ordered list)
  - *alt*: container rdf:Alt (alternatives)
- An instance of a class annotated with `ResourceClass` is mapped to a blank node including its properties. The mapper operates recursively.
- All other types are converted with the Java serializer to hexadecimal literals (base64).

Listing 6.1 shows an example of a resource class that is prepared for the object resource mapper: An instance of the class `Task` is mapped to a sub resource tree, which is shown in the Turtle syntax.

Future endeavors could add support for versioning to check the compatibility of a Java class and a resource tree.

6.5 Classes on Server-Side

Figure 6.2 shows a class diagram, including all important classes of Semantic MozartSpaces on server-side. It includes all associations between the listed classes. The gray-marked classes are a part of the native MozartSpaces. All other classes are a part of Semantic MozartSpaces. Table 6.1 lists the components from the chapter 5 with their associated Java classes.

SemanticBackend

This class is the root of the semantic back-end component and an implementation of the interface `StoreAPI`. It instantiates the following classes: `SemanticStore`, `SemanticContainerManager`, `SemanticEntryCoordinationDataMap`, and `SemanticContainer`.

Each instance of the class `SemanticEntryCoordinationDataMap` needs a unique label inside its container. Commonly, the label should be the name of the coordinator. The attachment of the coordination data to the resource tree of an entry requires that this label be a URI. For all labels that are not a valid URI, the class `SemanticBackend` creates a URI by concatenating the path `http://xvsm.org/semantic/coordinator/` and the encoded label.

```
@ResourceClass(defaultNamespace = "http://xvsm.org/tasks#")
public class Task implements Serializable {
  public Task() {
    super();
  }

  @ResourceProperty(name = "hasName")
  private String name;

  @ResourceProperty(name = "requiresSkill")
  private Set<URI> requiredSkills;

  // placeholder for public getters and setters
}
...
SubResourceTree subTree = ObjectResourceMapper.
        getInstance().mapToSubResourceTree(task);
```

sample resource class

```
[ a <http://xvsm.org/tasks#Task> ;
  <http://xvsm.org/tasks#hasName> "Task 1" ;
  <http://xvsm.org/tasks#requiresSkill>
        <http://xvsm.org/tasks#cutting> ,
        <http://xvsm.org/tasks#burning>
] .
```

sample RDF graph

Listing 6.1: Example for using the object resource mapper

The implementation of the function `createPersistentTransaction` of the interface `PersistenceBackend` does nothing. In the semantic back-end, transactional log-items are created on-demand where they are needed. The interfaces `StoredMap` and `OrderedLongSet` of the old persistence layer are not supported.

SemanticContainerManager

This class implements the interface `StoredContainerMap` of the Store API. It is responsible for persisting container descriptors, which are stored as resource trees.

This class adds log-items to MozartSpaces transactions for finishing container descriptor updates. During commit, write-locks are removed and the remove of container descriptors is executed. During rollback, container descriptors with write-locks are removed.

This class is used for persisting all container descriptors by the class `DefaultContainer-`

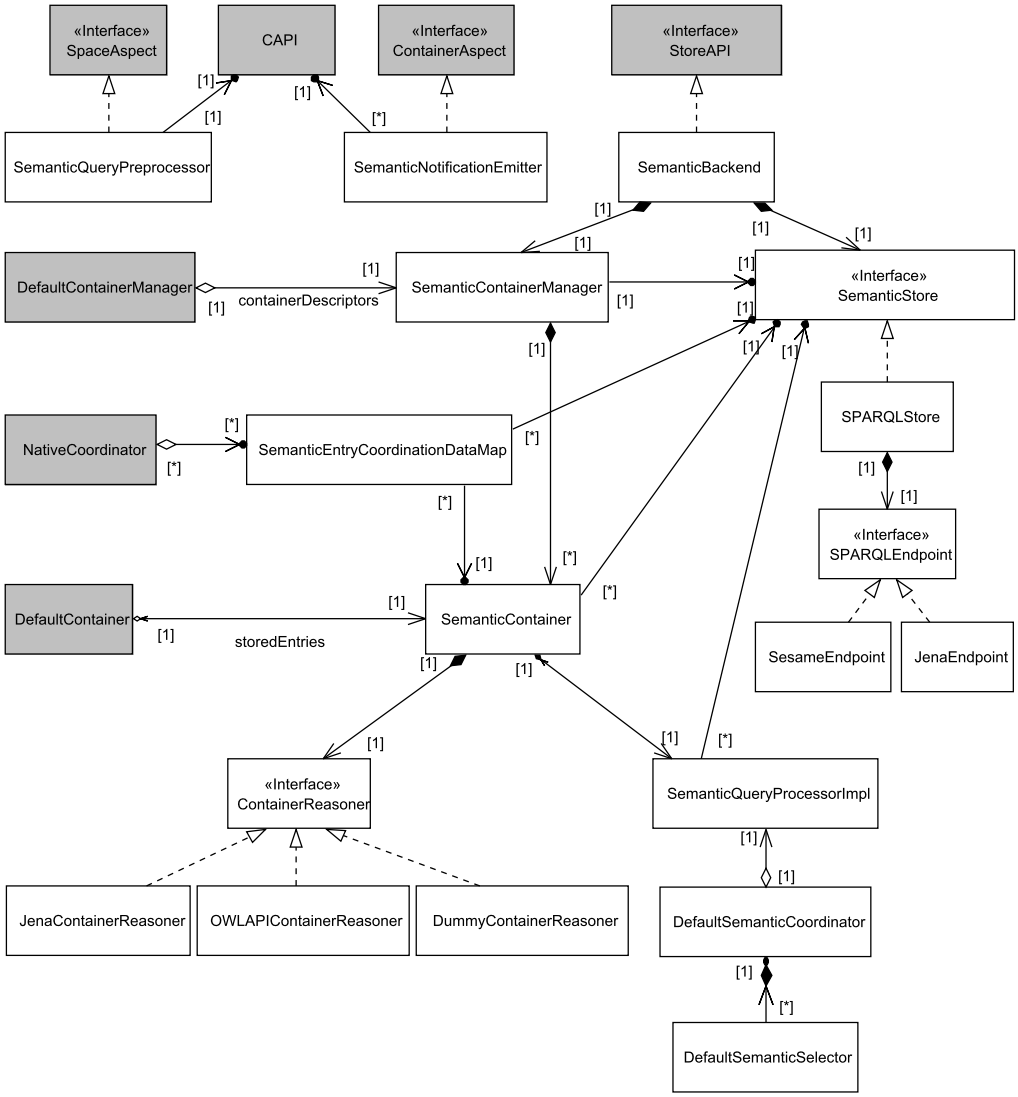

Figure 6.2: Class diagram (server-side)

`Manager`. During instantiation, `SemanticContainerManager` restores container descriptors, removes uncommitted descriptors, and removes temporary named graphs from the last MozartSpaces session. Therefore, it has to call the instance of the class `SemanticStore` several times. All data are stored in the named graph `xvsm:core-data`.

Internally the semantic container manager uses two Java maps for storing container descriptors and semantic containers. In both maps container IDs are used as keys.

Component	*Classes*
query pre-processor	`QueryPreProcessor`
notification emitter	`NotificationEmitter`

semantic coordinator	`SemanticQueryProcessorImpl`
	`DefaultSemanticCoordinator`
	`DefaultSemanticSelector`

semantic back-end	`SemanticBackend`
	`SemanticContainerManager`
	`SemanticContainer`
	`StoredEntryCoordinationDataMap`

triplestore	`SemanticStore`
	`SPARQLStore`
	`SPARQLEndpoint`
	`SesameEndpoint`
	`JenaEndpoint`

reasoner	`ContainerReasoner`
	`JenaContainerReasoner`
	`DummyContainerReasoner`

Table 6.1: Component membership

SemanticContainer

This class implements the interface `StoredEntryMap` of the Store API. It owns and creates an instance of the interfaces `ContainerReasoner` and `SemanticQueryProcessor`.

The class `DefaultContainer` uses the interface `StoredEntryMap` to manage and to persist all its entries. All entries are of the type `LazyNativeEntry` that have a weak reference to their values. The value of an entry is loaded on demand.

During instantiation of `SemanticContainer`, all entries of the container are restored from the triplestore. As an internal index structure a Java map with all the entries is used.

`SemanticContainer` collects all coordination data and combines them with the value of their entry. Then it stores them as resource trees with `SemanticStore` in the triplestore. Each container owns three named graphs: one for entries (raw data), one for the inferred data of the entries, and one for the container ontology.

Before an entry is written to the triplestore, the reasoner is used to produce the inferred data of the entry. To each MozartSpaces transaction that includes an entry write, a log-item is added. During commit, the write-locks of the entries are removed in the triplestore. During rollback, the uncommitted entries are removed from the triplestore.

Depending on the semantic configuration properties, inferred data are not only attached to the entries in the triplestore, but also to the value of the MozartSpaces entries (instances of the interface `NativeEntry`) that are returned to the user.

`SemanticContainer` informs its instance of `ContainerReasoner` about new and deleted ontology entries, and stores in the triplestore the updated ontology graph returned by the

reasoner. At the moment, the reasoner is called during commit, inside the transactional log-item. In the case of an invalid updated ontology, or an error during the update of the ontology graph in the triplestore, this can cause an inconsistent container state. In future works, the ontology update should be done in a pre-commit aspect. The best approach would be to extend MozartSpaces log-items with a pre-commit function.

SemanticSerializer

This class is used by the classes `SemanticContainer` and `SemanticEntryCoordinationDataMap`. It is needed to map entry value or entry coordination data of the type `Serializable` to a sub resource tree and visa versa. This is done by the functions `mapToSubTree` and `mapToSerializable`.

In the function `mapToSubTree` used to map an object to a sub resource tree, the way objects are treated depend on their data type:

- An instance of `SerializedSubResourceTree` is deserialized.
- A URI is used as the root and the leaf at once of the new sub resource tree.
- Literals are converted to their XML Schema equivalent, and treated as URI above.
- An instance of a `ResourceClass` annotated class is mapped to a sub resource tree by the class `ObjectResourceMapper`. A flag is added to the sub resource tree that it was mapped on server-side. In the function `mapToSerializable`, only sub resource trees with this flag are mapped by the object resource mapper to an object of their `ResourceClass` annotated class.
- Other objects are serialized with the Java serializer. The byte array is encoded with base64, and treated as a literal.

ContainerReasoner

An implementation of this interface manages the ontology of its container, and produces inferred data for entries.

The class `UpdateTask` is used for adding or removing an ontology entry to the container's ontology. The function `update` should be called with all update tasks of a MozartSpaces transaction at once. This function returns the new ontology, including its inferred data. If the ontology is empty, it will return `null`.

At the moment there exist three implementations of the container reasoner:

- `DummyContainerReasoner` does nothing. It can be used for container that needs no reasoning.
- `JenaContainerReasoner` uses the reasoner of Jena that offers only the OWL 1 support.

- `OWLAPIContainerReasoner` uses the OWL API, which has connectors to many reasoners. At the moment the reasoner Hermit is used. The OWL API does not support blank nodes in an RDF graph. Before reasoning, all blank nodes in the resource tree of an entry are mapped to temporary URIs. After reasoning, they are mapped back to blank nodes. Because the OWL API has its own data model for the RDF graphs, it has to be converted to the Jena model.

SemanticEntryCoordinationDataMap

This class implements the interface `StoredEntryCoordinationDataMap` of the Store API. An instance of this class manages the entry coordination data of a coordinator in a container. The coordination data are attached to resource trees of entries with the property label that is a URI.

In the function `put`, the coordination data are forwarded to the class `SemanticContainer`, which stores them at once within their entry into the triplestore.

The function `get` returns the coordination data of a specific entry. The interface `PersistenceCache` is used as the cache for coordination data. If the requested data are not in the cache, they are read from the triplestore with the Semantic Store API.

The function `remove` removes data only from the map-internal data structure. Coordination data cannot be removed separately from their entries. They are removed from the triplestore by the class `SemanticContainer`. In future works, the function `remove` should be triggered by `SemanticContainer`, and not by the coordinator.

A set with all entry IDs with coordination data of that map is used as an internal index structure.

SemanticQueryProcessor

This interface consists of the single function `query` for executing semantic queries. `SemanticQueryProcessorImpl` is the implementation of this interface.

The execution is done in four steps:

1. The lists of context entries are stored in temporary named graphs in the triplestore. In future works, inferred data could be optionally produced for context entries and inserted into the temporary graphs, too.

2. The SPARQL SELECT query string is generated (described in section 5.7). It includes variable bindings for the context entry graphs. Preselected entries of the predecessor selectors are respected in the SPARQL query.

3. The query string is executed, and as a result, an entry iterator is created. The result also includes the uncommitted entries of all transactions. The isolation manager of MozartSpaces has to filter the visible entries. This can be done by the class `FilterEntryIterator` in combination with an instance of `AccessibilityEntryFilter`. For the selectors with a predecessor, another additional accessibility check is not necessary. The needed size

of the query result varies depending on the transactional locks of the entries. The semantic selector asks for the next result until the selector count is reached.

4. After query executions, the temporary named graphs are destroyed. This is triggered by the function `close` of the entry iterator.

DefaultSemanticCoordinator

This class is a custom, implicit MozartSpaces coordinator implementation. It does not need explicit coordination data. Only one instance of this class per container is possible. It makes no difference if the semantic coordinator is obligatory or optional. It implements the interfaces `PersistentCoordinator`, and `ImplicitNativeCoordinator`.

The main task of this class is to offer the class `DefaultSemanticSelector` a reference to the query processor (`SemanticQueryProcessor`). For getting this reference, a workaround was necessary. MozartSpaces does not offer any interface between a coordinator and a Store API implementation. A pseudo entry coordination data map is created. The semantic back-end returns an instance of the class `SemanticQueryProcessorGetter`. This class acts only as a wrapper for the query processor, and has no other functionality. This workaround is hidden in the semantic server API (`SemanticServerAPI`).

The semantic server API also makes it possible that other custom coordinators can use the semantic query functionalities.

The second task of the semantic coordinator is to attach the coordinator-independent metadata (`EntryMetadata`) to the root level of the resource tree of an entry. This is done by special entry coordination data maps, which are initialized on-demand. If the label of the map is already in use by another coordinator, then an exception `CoordinationDataNameAlreadyInUseException` is thrown.

DefaultSemanticSelector

This class is an implementation of the interface `NativeSelector`. It selects a list of entries matching a semantic query. For every read request that includes an instance of `SemanticSelector` (the model class for selector data), an instance of `DefaultSemanticSelector` is created. An instance of `SemanticSelector` consists of a semantic query (`SemanticQuery`), a list of context entries, and a selector count.

For executing a semantic query the `SemanticQueryProcessor` is called. As a result, it produces an entry iterator. The semantic server API encapsulates the execution of the query, and the isolation check. It asks for new entries until the selector count is reached. By using this API, other custom sectors can also use this functionality.

Section 5.7 describes the strategy for calling the predecessor selector. The problem with the current implementation is that the execution of a semantic query is expensive. In most cases, multiple calls of the predecessor are faster than one extra call to the semantic query processor. Future work should optimize query processing by implementing a preloading of the semantic query.

In future works, to obtain a better streaming strategy, the interface of MozartSpaces selectors should be extended to include a ”close” function. Inside this function, the selector count can be checked, and the entry iterator of the query processor can be closed.

SemanticStore

This interface offers functions for reading, searching, and updating resource trees. It acts as an adapter between the triplestore and the other classes of the semantic back-end.

All functions run in a single triplestore transaction, which is committed at the end of the functions.

Two implementations of the `SemanticStore` already exist:

- `SPARQLStore` bunches all SPARQL queries described in chapter 5. It is independent of any triplestore implementation. For accessing a triplestore, it uses the interface `SPARQLEndpoint`. This interface includes a function for every SPARQL query form (SELECT, ASK, CONSTRUCT, and UPDATE). However, a triplestore implementation must support version 1.1 of SPARQL.

  The classes `JenaEndpoint` and `SesameEndpoint` are the implementations of `SPARQLEndpoint`. To prevent concurrency problems, both have their own read-write-locks. Both endpoints support in-memory and persistence strategy. In the semantic configuration properties, this strategy can be defined for a Semantic MozartSpaces runtime. Jena and Sesame also offer support for SQL databases, which is slower than the native storage. In future works, it would be easy to add this support to endpoint implementations.

  For entry insertion, including the inferred data, a special graph serializer is necessary to generate the SPARQL `CONSTRUCT` queries. As defined in section 5.8, inferred data are stored in a separate named graph. The same blank node labels have to be used for the raw data and the inferred data of an entry. Therefore, the class `BNTurtleWriter`, an adopted Jena Turtle serializer, was implemented. It is a Turtle writer without the blank node abbreviations. It always uses the same labels for the same blank nodes for serializing different graphs.

- `JenaStore` uses the native API of Jena. This achieves a better performance than `SPARQLStore` with Jena as endpoint.

The class `SemanticBackend` instantiates the implementation of the `SemanticStore` that is declared in the semantic configuration properties. At the moment, only one semantic store per MozartSpaces core can exist. In future works, this can be changed to support semantic containers with different persistence strategies on the same core. One triplestore instance per container could achieve a better concurrency. This optimization would make the guarantee of atomic transactions that include different containers complicated.

SemanticQueryPreprocessor

This class is a space aspect for the following entry operations: pre-read, pre-take, and pre-delete.

For each `SemanticSelector` instance, this class processes the following three tasks:

- Optionally, it loads a stored query. A semantic selector has a list of selectors to retrieve a stored query. In the case this list is not empty, a read request with these selectors is executed. The read entry query is added to the semantic selector.

- It validates the semantic query against the rules defined in section 5.7. The class `QueryValidation` implements this validation functionality. It uses Jena ARQ to walk through all elements of a SPARQL query. In the case of an invalid query, an `IllegalQueryException` is thrown.

- It executes sub-queries and adds their results as new context entry lists to the main entry query. A sub-query is a read request with a list of `SerializedSubResourceTree` as the result.

SemanticNotificationEmitter

This class is a container aspect for the following entry operations: post-read, -take, -delete, and -write. For subscriptions that are interested in a processed operation and its entry, a notification is sent. This aspect is not loaded by default, because it causes performance loss.

This class implements the design presented in section 5.9. One change had to be done. In MozartSpaces, the read and the take aspects get only the value of an entry, not the complete entry. For that reason, the OWL class defined in a subscription refers to the types of an entry value, instead of the types of an entry. Therefore, in order to use the inferred OWL classes of the entry values, in the semantic configuration properties, the addition of the inferred data to the entries has to be enabled.

The user writes into the container that should be observed subscriptions as entries with instances of `SemanticSubscritpion` as values.

For every entry operation the notification emitter tries to read subscription entries that are interested in it. This read operation does not trigger any aspects, because it is done on the CAPI-3 level. The semantic query that is used for this read operation is described in the section 5.9. For every found subscription, a notification entry is written into the notification container that is declared in the subscription entry. This write operation is done inside the transaction of the notified operation. The notification entry is only visible after such transaction is committed. Within the rollback of this transaction written notification entries are removed.

A notification container must have a specific label coordinator. The name of this coordinator must be ”semanticSubscriptionID-label” (stored in the constant SemanticNotification.SUBSCRIPTION_ID_LABEL). All written notification entries are labeled with their subscription IDs.

MozartSpaces transactions are only valid on a single core. For this reason, the notification container has to be on the same core as the observed container. In future works, notification entries should be stored in a special, local container. A post-commit aspect should rout the notification entries to their real target notification containers.

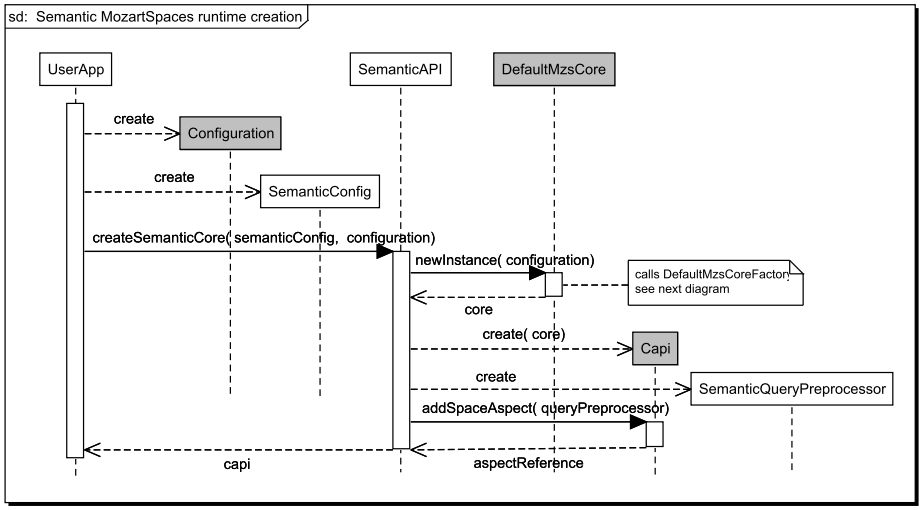

Figure 6.3: Semantic MozartSpaces runtime creation (part 1)

6.6 Operations

This section describes all general operations of Semantic MozartSpaces: container creation, container destroy, container restore, entry write, entry read, and entry remove. The focus of this section lies on the Semantic MozartSpaces specifics. The descriptions of the operations do not include all steps that are processed by native MozartSpaces.

Each operation is described by sequence diagrams. Lifelines represent Java classes. The gray-marked classes are a part of the native MozartSpaces. All other classes are a part of the Semantic MozartSpaces implementation. The diagrams are simplified, and contain only the most important parameters. To save space, some diagrams use an asterisk (”*”) as an abbreviation for the word ”Semantic”. The phrase ”*Container”, for example, stands for the class ”SemanticContainer”. The parameter ”void” of the return messages is not explicitly listed.

Initialize Semantic MozartSpaces

Figures 6.3 and 6.4 show the creation of a new Semantic MozartSpaces runtime. The function `createSemanticCore` of the class `SemanticAPI` hides the necessary tasks. It handles the correct initialization and configuration for a Semantic MozartSpaces runtime.

In the semantic configuration properties (`SemanticConfig`), the user can define the Semantic Store API implementation (SPARQLStore or JenaStore), the reasoner implementation, the persistency strategy (in-memory or persistent) and the addition of inferred data to the read entries.

A single instance of the class `SemanticBackend` is created by the constructor of the class `PersistenceContext`. It creates the Semantic Store and an instance of the class `SemanticContainerManager`. In the constructor of this container manager, the cleanup of the last session and restore of persisted containers is performed. The restore is finished in

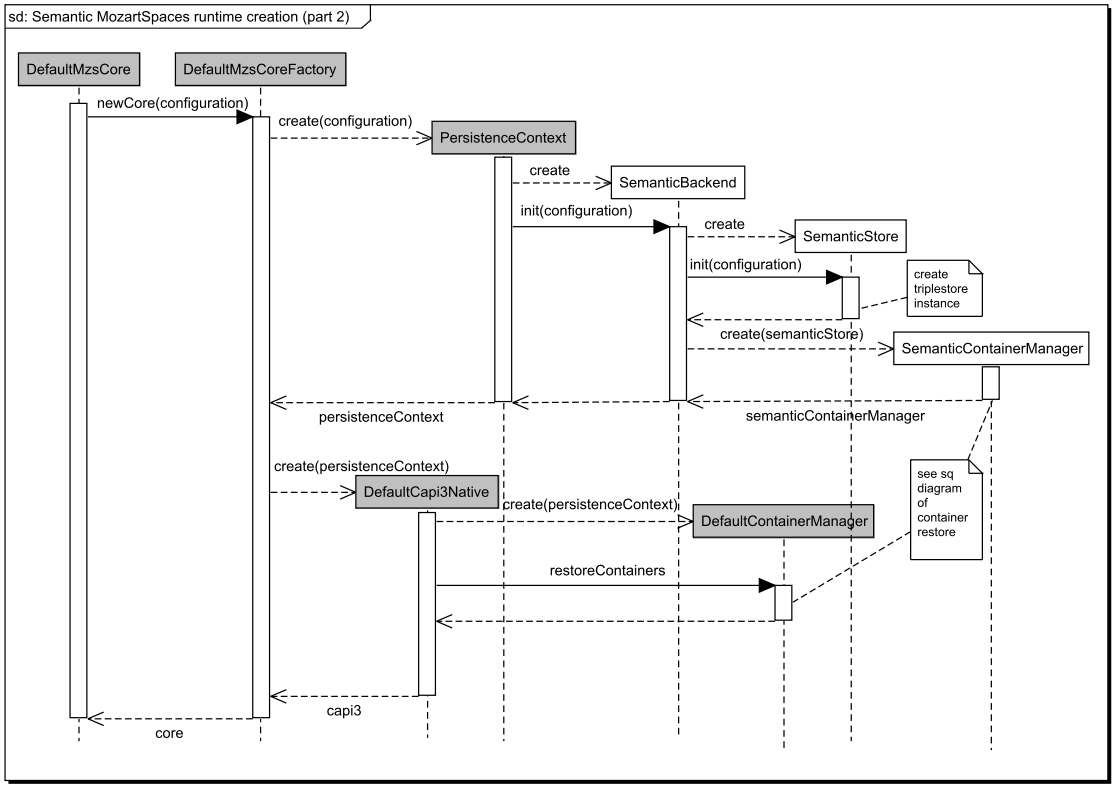

Figure 6.4: Semantic MozartSpaces runtime creation (part 2)

the function `restoreContainers` of `SemanticContainerManager`. The subsection about the container restore contains more details on this subject.

At the end of the initialization, an instance of `SemanticQueryPreprocessorImpl` is added as a space aspect to the MozartSpaces runtime.

Container Creation

Each container must include `SemanticCoordinator` as an optional coordinator. Otherwise, the container could not be used for semantic queries. In a future version of Semantic MozartSpaces, `SemanticCoordinator` can be attached automatically to each container by a pre-create-container aspect.

Figure 6.5 describes the procedure of creating a semantic container on server-side. The user request on the CAPI-4 level forces the `CAPI3` instance to call `DefaultContainerManager` for creating a new container (`DefaultContainer`).

An instance of the class `DefaultContainer` uses an instance of the interface `StoredEntryMap` for storing all entries of the container. The class `SemanticContainer` implements this interface. The constructor of `DefaultContainer` calls the function `createStore-`

dEntryMap of the class PersistenceContext, which, in turn, calls the class SemanticBackend to create a new instance of SemanticContainer. In this step, instances of the interfaces ContainerReasoner and SemanticQueryProcessor (query processor) are also initialized.

In the next step, the semantic coordinator (DefaultSemanticCoordinator) gets a reference of the query processor of its container. In MozartSpaces, a coordinator has no direct interface to communicate with the Store API. A workaround is needed to retrieve the instance of the query processor. The function initPersistence in DefaultSemanticCoordinator calls the function getQueryProcessor of the class SemanticServerAPI. This function forces the class SemanticBackend to create a pseudo instance of the interface StoredEntryCoordinationDataMap that acts only as a wrapper for the query processor.

The instance of the class DefaultContainerManager hands the container descriptor (PersistentContainerDescriptor) over to the instance of SemanticContainerManger that stores it with a write-lock in the named graph xvsm:core-data of the triplestore. SemanticStore handles this storage. An extra log-item is attached to the transaction for handling commit and rollback. During commit, the write-lock is removed of the stored container descriptors. During rollback, the container descriptor and all named graphs of the containers are removed.

Container Destroy

By calling the function destroyContainer of an instance of the class Capi (the API of native MozartSpaces), a container destroy task is created and sent to the runtime of the container. Figure 6.6 shows the handling of such a request in the container-hosting MozartSpaces runtime.

The function remove of SemanticContainerManager creates a log-item for the transaction of the request. During commit, this log-item removes all named graphs of the container and its descriptor in the named graph xvsm:core-data.

Container Restore

At every start of a Semantic MozartSpaces runtime all persisted containers are restored. The sequence diagram split to the figure 6.7 and 6.8 gives an overview of restoring containers.

In the constructor of SemanticContainerManager, the temporary named graphs of unfinished semantic queries of the last session are dropped. All container descriptors are read from the named graph xvsm:core-data. The descriptors with a write-lock are removed, including all named graphs of their containers.

In the function retoreContainers of DefaultContainerManager, all container descriptors are received from the instance of SemanticContainerManager. For each descriptor a new instance of DefaultContainer is created. Its constructor initializes SemanticContainer that restores the entries from the triplestore and rebuilds the container ontology. DefaultContainer iterates over all entries to find the highest ID. The entry ID counter is set to this ID. It is responsible for assigning IDs to the new entries.

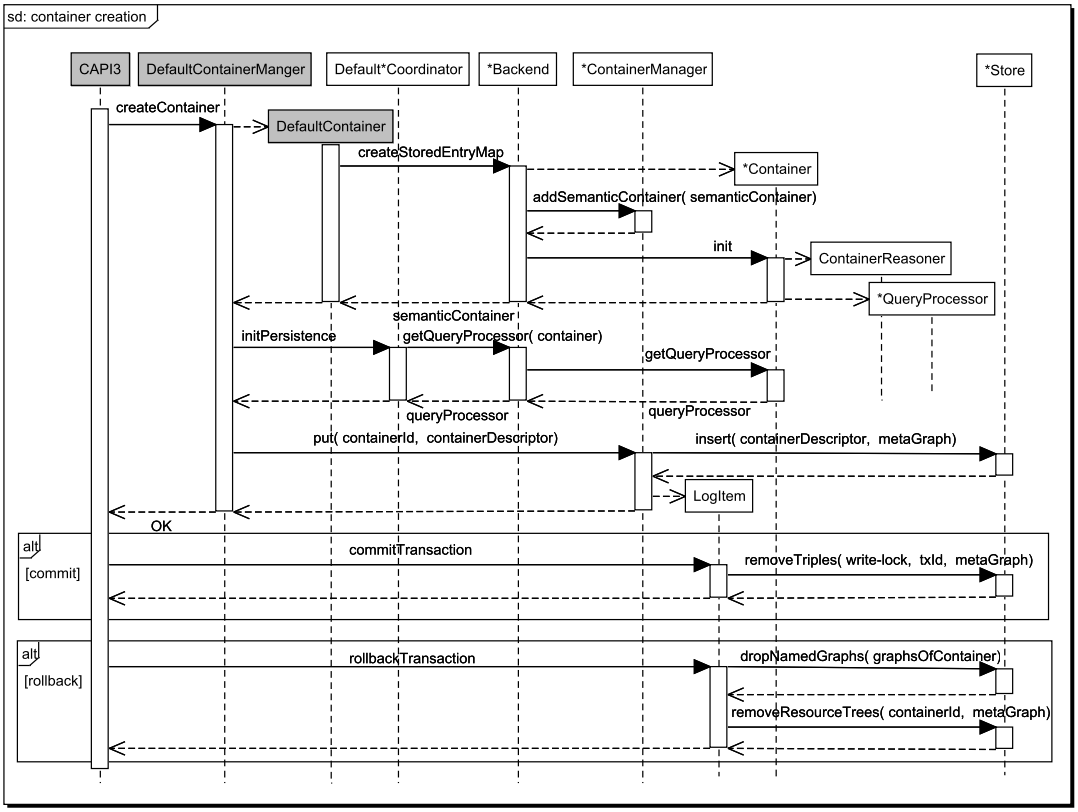

Figure 6.5: Container creation (server-side)

Entry Write

This subsection describes the operation of storing a semantic entry in a container. Figure 6.9 shows the insertion of a Java object as a semantic entry in a container. The class of this object must be prepared for the object resource mapper. The function `serialzeEntryValue` of `SemanticAPI` converts this object to a serialized sub resource tree. A new instance of `Entry` is created with the sub tree as its value. This entry is written into a container by using an instance of the class `Capi`. A serialized sub resource tree can be also produced by using other tools, or by hand.

Figure 6.10 shows the processing of a write request in the container-hosting MozartSpaces runtime. The function `executeWriteOperation` of the class `DefaultContainer` is responsible for entry write requests on the container level. First, the inserted entry is registered at all coordinators. In the case where a coordinator persists its coordination data, it calls the function `put` of its instance of `SemanticCoordinationDataMap`. The coordination data are mapped to a sub resource tree and are handed over to `SemanticContainer`.

Next, the entry is stored by `SemanticContainer` in the triplestore. If a log-item for the container does not already exist for the transaction, it will then be created. This log-item

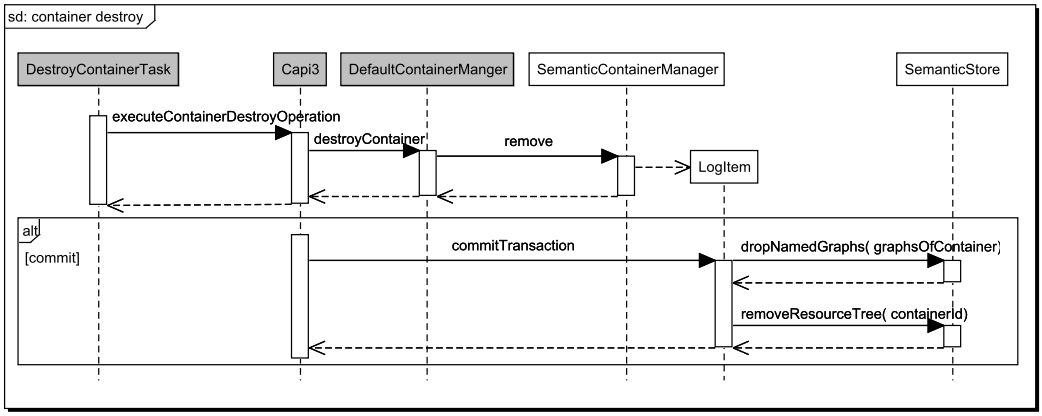

Figure 6.6: Container destroy (server-side)

handles the final tasks during commit or rollback. If the value of an entry is an instance of `OntologyEntryValue`, a new ontology update task is added to the log-item. A resource tree for an entry is build of its value and its coordination data. The container reasoner produces inferred data for this resource tree. The resource tree with a write-lock and its inferred data are stored in the triplestore by `SemanticStore`. The raw data are inserted into the named graph of the container (e.g. `container:1` for container with ID "1") and the inferred data into the named graph for the inferred data of the container (e.g. `container:1/inferred`).

The function `storeEntries` of `SemanticContainer` is called by `DefaultContainer` after the access control check. Thus, the use of the semantic coordinator as a filter for access control is in this implementation not possible.

Figure 6.11 shows the rollback of the sub-transaction of a write request. For each written entry an own instance of the class `DefaultWriteLogItem` exists. For all written entries in the sub-transaction, the instance of the class `TransactionalTask` triggers the call to the function `purgeEntry` of the `SemanticContainer`. The resource tree of an entry is removed from the triplestore.

The processing of the commit of a write request is described by figure 6.12. The log-item of the container removes the write-lock of the transaction from all entries. This is done by removing all triples in the named graph of the container with `sxvsm:hasWriteLock` as the predicate, and the ID of the transaction as the object. In the case the update task list is not empty, the function `update` of `ContainerReasoner` is called. The old ontology graph of the container is dropped and the new graph produced by the reasoner is inserted into the triplestore.

The rollback of a write request is shown in figure 6.13. `DefaultWriteLogItem` triggers the call of the function `purgeEntry` of `SemanticEntry`. In this case, this function does nothing. All entries written in the transaction are removed at once by the log-item of `SemanticContainer`. The log-item calls `SemanticStore` to remove all resource trees of the written entries.

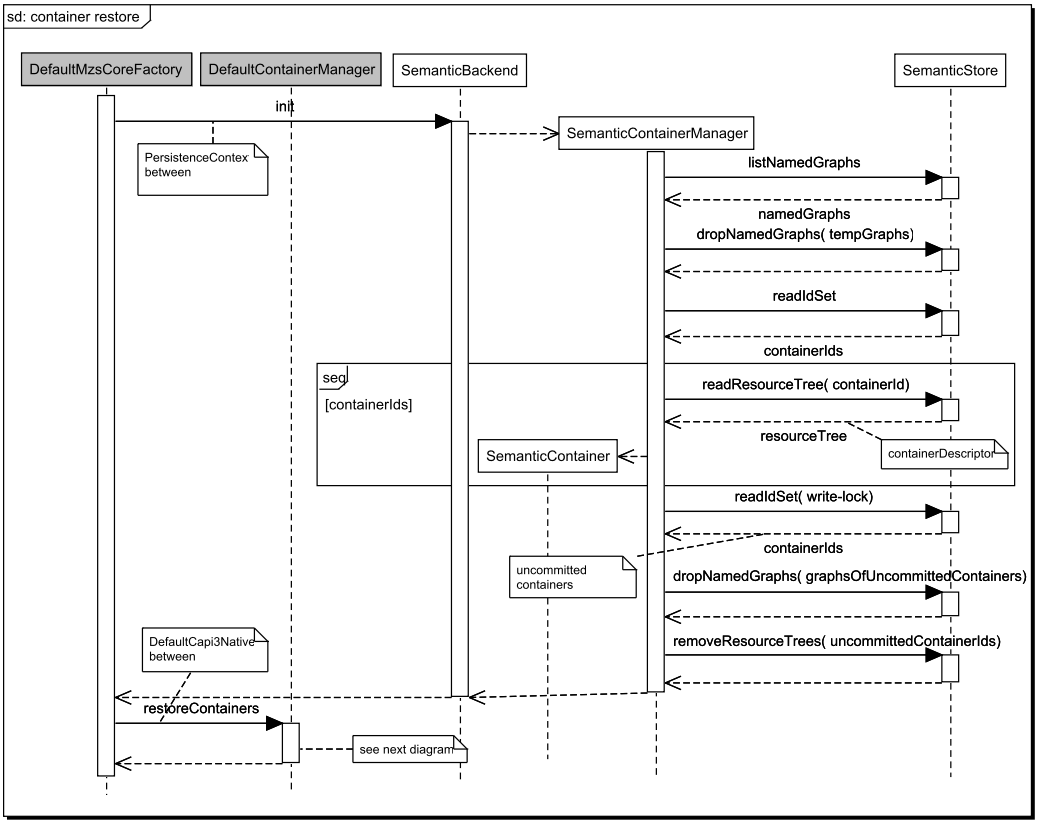

Figure 6.7: Container restore (server-side; part 1)

Semantic Entry Read

Figure 6.14 shows a client application that reads semantic entries by using a semantic query. With the help of the class `SemanticAPI` a new semantic query is created. The query is wrapped with an instance of `SemanticSelector`. Then the read operation, including the selector, is executed by calling an instance of the class `Capi`. The result is a list of serializable objects. In the case of semantic entries, these are the instances of `SerializedSubResourceTree`. If for the sub resource trees a compatible Java class is registered in the object resource mapper, `SemanticAPI` can map them to their equivalent plain Java objects.

The processing of the read request in the container-hosting MozartSpaces runtime is shown in figure 6.15. The pre-aspect `SemanticQueryPreprocessor` has three tasks. Optionally, it reads a stored query from the container with the selectors defined in `SemanticSelector`. This is done by a call to `Capi3` that does not trigger any aspect. The query of `SemanticSelector` is validated against the rules presented in section 5.7. In the case of a rule violation, an exception (`IllegalQueryException`) is thrown. The sub-queries of `SemanticSelector` are executed. Sub-queries are read requests with a list of `SerializedSubResourceTree` as the result. The results are attached to the instance of `SemanticSelector` as new context entry

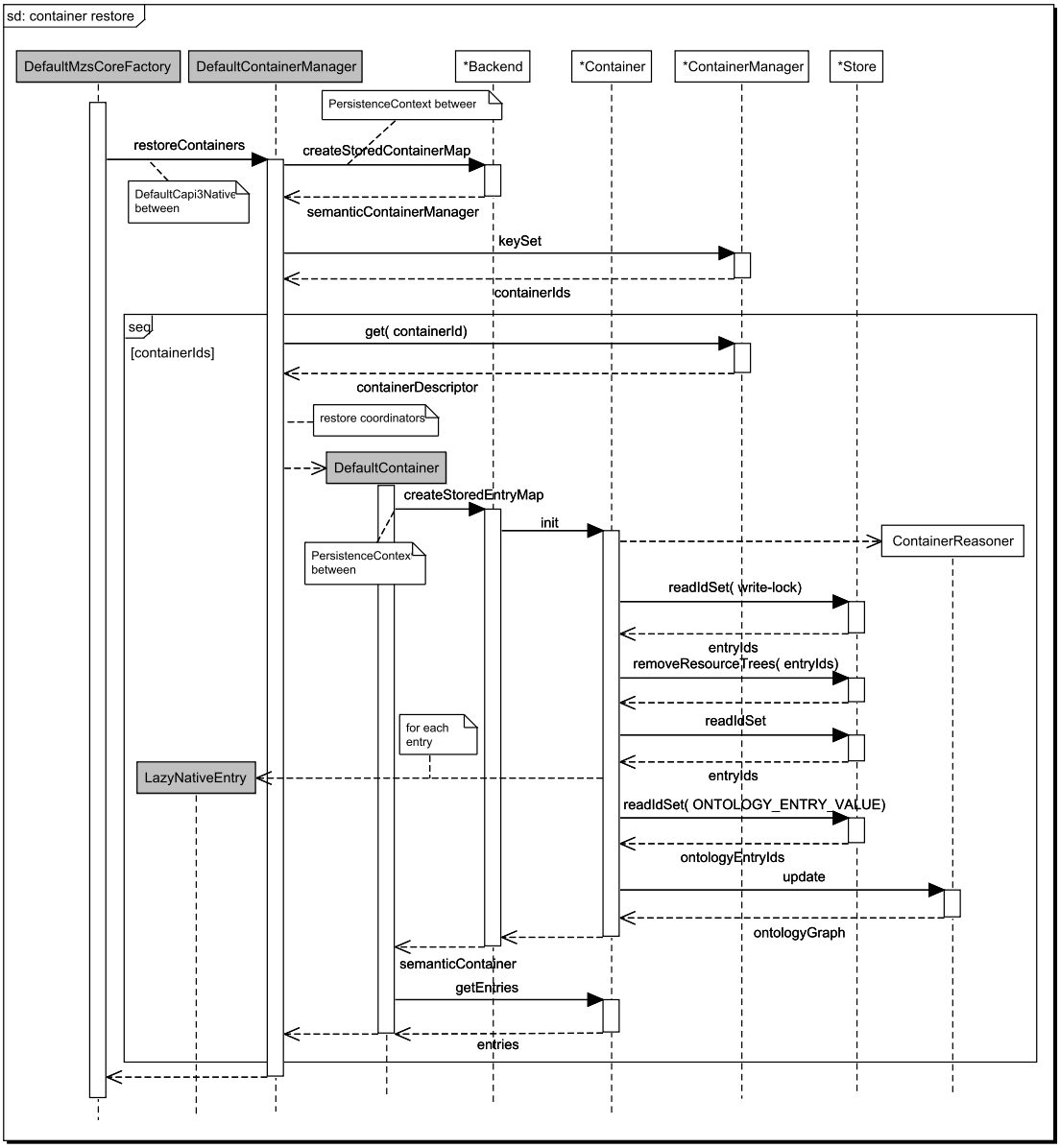

Figure 6.8: Container restore (server-side; part 2)

lists.

After the pre-aspects, the selectors are executed. In the sequence diagram, the instance of the class `DefaultSemanticSelector` is the only selector. Figure 6.16 shows the details of this part of the execution. `SemanticQueryProcessorImpl` handles the execution of the semantic query. At first, the context entry lists are stored in the separate, temporary named graphs.

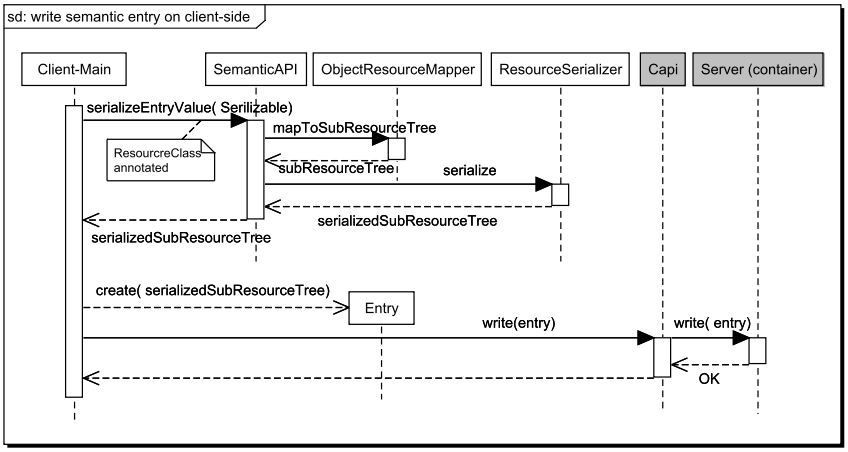

Figure 6.9: Entry write (client-side)

A SPARQL query is generated from the semantic query. It is executed in the triplestore. The result is an iterator of the entries IDs. `SemanticServerAPI` calls the iterator until the selector count is reached. The accessibility of each entry is verified by the isolation manager. Invisible entries are not counted in reaching the selector count. At the end, the triplestore operation is closed.

An instance of `SemanticContainer` contains a map with all entries (`NativeEntry`) of its container. The values of these entries are weak references. If an entry object is read that does not include its value anymore, then its value is loaded again from the triplestore.

After each processed entry request (read, take, and write) the optional aspect `SemanticNotificationEmitter` is called.

Entry Remove

Entry delete requests are processed during the commit of their transactions. Before commit, the entries are only hidden by the MozartSpaces isolation manager. The sequence diagram in figure 6.17 describes this operation in the container-hosting MozartSpaces runtime. In the function `purgeEntry` of `SemanticContainer`, the resource tree of the requested entry is removed from the triplestore. If an entry is registered as an ontology entry in the container reasoner, then an ontology remove task will be added to the container's log-item. In the commit processing of the log-item, all ontology update tasks (write and remove) are processed at once. The ontology graph is updated in the triplestore.

6.7 Configuration

For correctly running Semantic MozartSpaces, the configuration properties (`org.mozartspaces.core.config.Configuration`) of the native MozartSpaces have to be extended with the semantic configuration. This should be done by the function

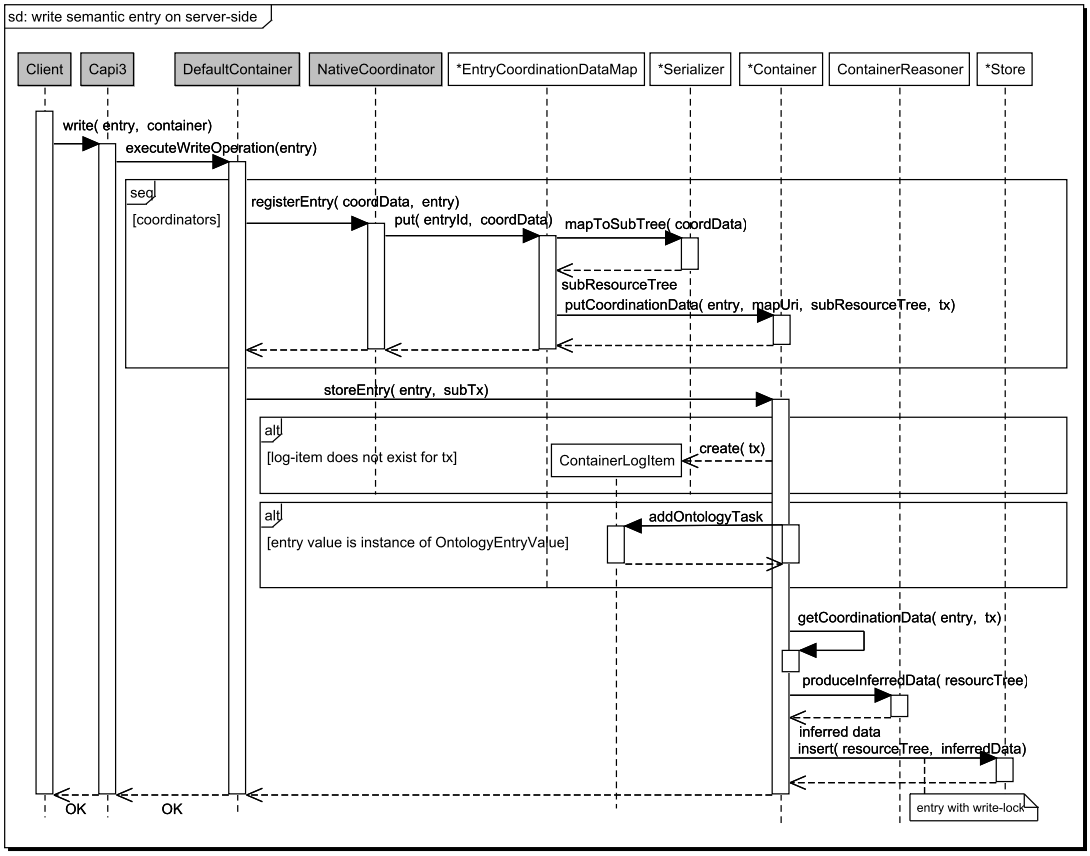

Figure 6.10: Entry write (server-side)

`setSemanticConfiguration` of `SemanticAPI`. The settings concern the persistence configuration of the properties. The class `SemanticBackend` is set as the persistence profile. An instance of `SemanticConfig` is set as persistence properties.

At the moment, container specific settings are not possible. The settings can be done only for a complete Semantic MozartSpaces runtime instance. These setting cannot be changed at run time.

The following properties can be set in the semantic configuration (`SemanticConfig`):

- `semanticStoreClass`: the implementation class of the Semantic Store API
- `sparqlEndpointClass`: the implementation class of `SPARQLEndpoint` (only relevant in combination with `SPARQLStore`)
- `containerReasonerClass`: the implementation class of `ContainerReasoner`
- `inferenceRule`: inference rules (RDFS or OWL) for the container reasoner (can be ignored by reasoner implementations)

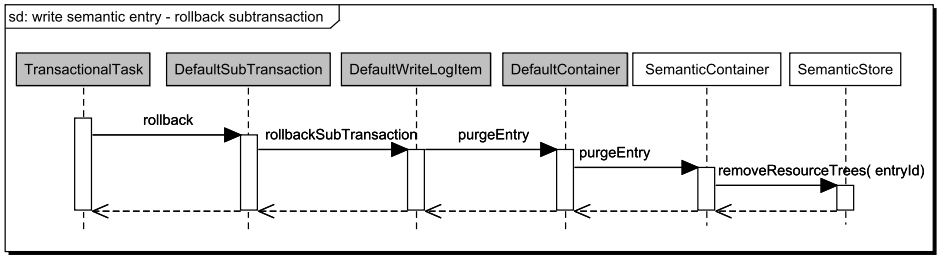

Figure 6.11: Rollback of sub-transaction of entry write request (server-side)

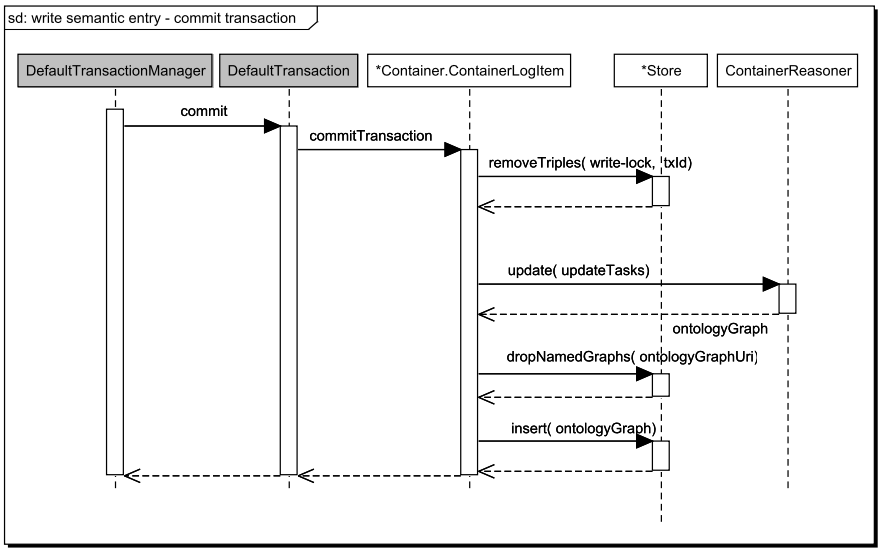

Figure 6.12: Commit of entry write request (server-side)

- `additionOfInferredData2Entries`: the option to add inferred data to entry values that are returned in read requests
- `persistenceStrategy`: The data of the triplestore can be stored only in-memory, or persisted to a hard-disk.
- `persistenceDirectory`: the directory to store the persistent data of the triplestore.

6.8 Build Tool

Maven was used as build tool for Semantic MozartSpaces. For an optimal dependency management, the implementation was split into five projects. Table 6.2 lists them.

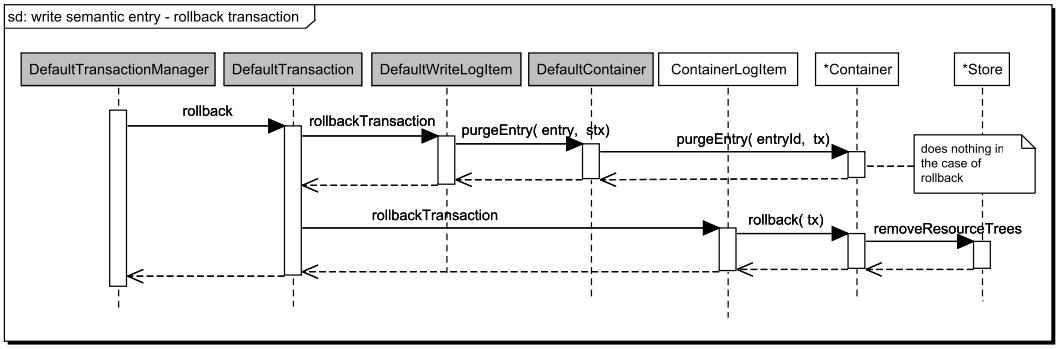

Figure 6.13: Rollback of entry write request (server-side)

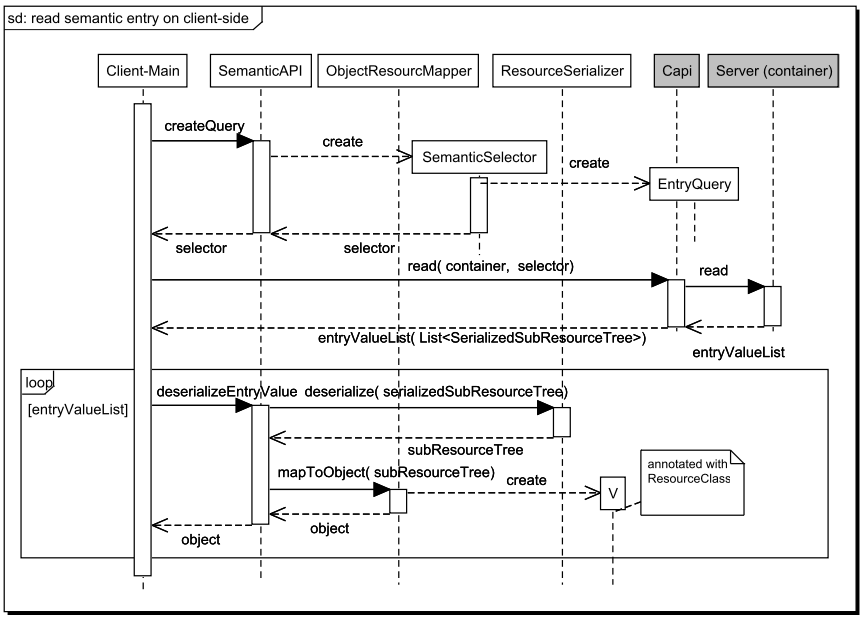

Figure 6.14: Entry read (client-side)

An extra repository is only necessary for retrieving MozartSpaces files: `http://www.mozartspaces.org/maven-snapshots`. All other dependencies are included in the main repository of Maven.

6.9 Summary

Semantic MozartSpaces implements the complete design described in the previous chapter, and provides the first proof of the functionality of the design. There are some small limitations in the implementation. Most of them could be eliminated by adopting the architecture of the native

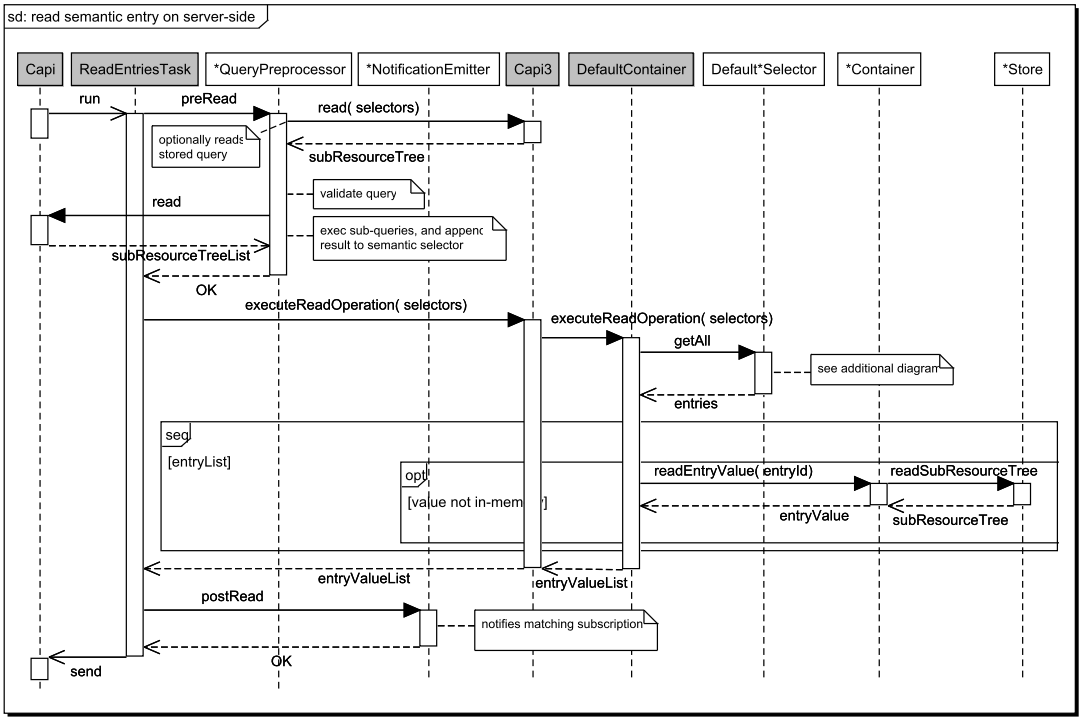

Figure 6.15: Entry read (server-side)

MozartSpaces. Future work can correct the following problems:

- Two semantic containers in the same runtime cannot have different persistence strategies (in-memory and persistence).
- The access control cannot use semantic coordinators for filtering.
- The user is not able to read coordination data (metadata) of the entries. This fact had also implications for the implementation of context entries and the semantic notification. Only the values of the entries can be used as context entries, and can be observed by the semantic subscriptions, instead of complete entries defined in the design.

Future works should improve the handling of transactions in Semantic MozartSpaces. Thus, a system crash during a commit cannot cause an inconsistent state in the persisted data model.

The semantic back-end was successfully tested with the integration tests of MozartSpaces. A JUnit test suite was created to test the correctness of Semantic MozartSpaces. The tests are included in the Maven project ”semanticspaces-test”.

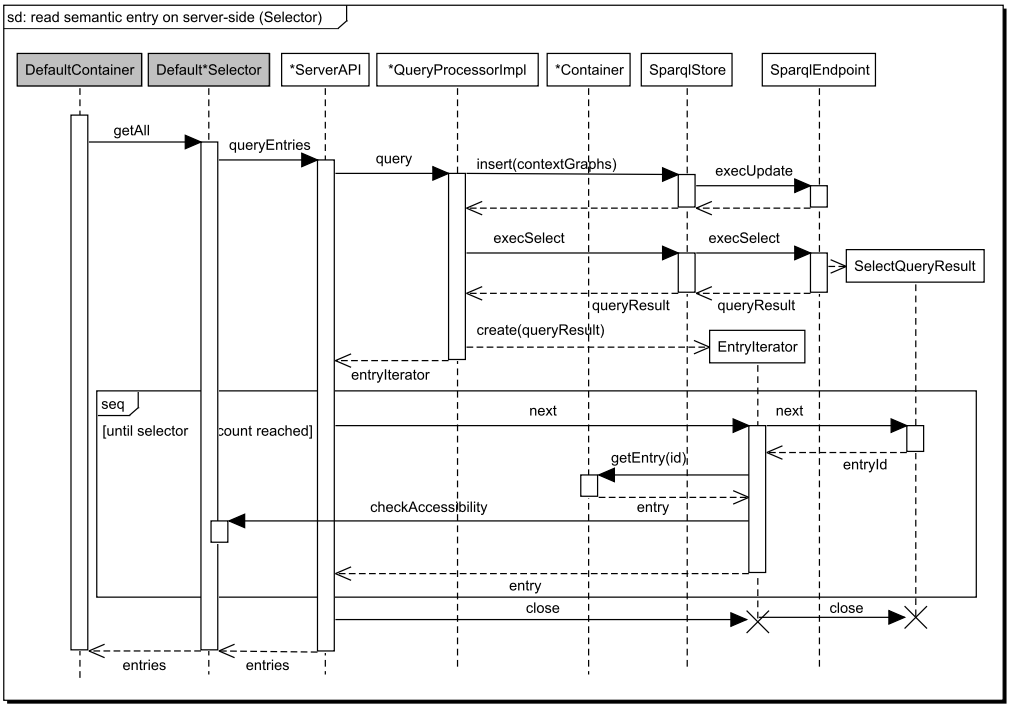

Figure 6.16: Entry read - selector (server-side; `SparqlStore` as Semantic Store)

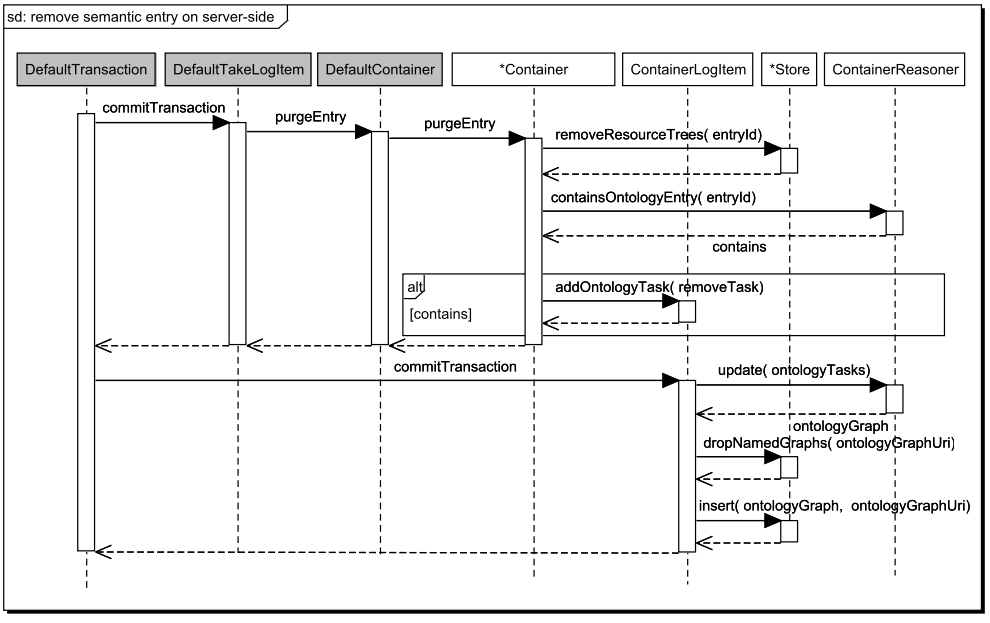

Figure 6.17: Entry remove (server-side)

Artifact ID	*Short Name*	*Classes*	*Dependencies*
semanticspaces-core	Core	*	mozartspaces-runtime, mozartspaces-core-cache, mozartspaces-notification, jena-arq
semanticspaces-jena	Jena	JenaStore, JenaEndpoint, JenaContainerReasoner	semanticspaces-core, jena-tdb
semanticspaces-sesame	Sesame	SesameEndpoint	semanticspaces-core, sesame-repository-sail, sesame-sail-nativerdf, sesame-sail-memory
semanticspaces-owlapi	OWL-API	OWLAPIContainer-Reasoner	semanticspaces-core, owlapi-distribution, org.semanticweb.hermit
semanticspaces-test	Test	*	semanticspaces-*, junit

Table 6.2: Maven Projects

CHAPTER 7

User Guide

This chapter is focused on user applications that use Semantic MozartSpaces.

A user has to deal with the semantic API (`SemanticAPI`) and model classes that were presented in section 6.3. In this thesis, the native MozartSpaces API was not changed. The basic knowledge about MozartSpaces is a precondition for understanding this chapter. The MozartSpaces website [14] is a good starting point. It provides a tutorial and code examples.

The description on how to prepare classes that are compatible with the object resource mapper are presented in section 6.4.

All code examples in this chapter are connected with each other, and can be found in the class `org.mozartspaces.semantic.UserGuide` in the Maven project "semanticspaces-test".

Semantic MozartSpaces offers an ontology support. During the execution of a write operation in the container-hosting MozartSpaces runtime, semantic entries are enriched with the inferred data. With the semantic selector one can define SPARQL queries to declare in which entries one is interested. This selector can use the inferred data.

7.1 Instantiation

With the function `createSemanticCore` of `SemanticAPI`, a new runtime instance of Semantic MozartSpaces is created. This function needs two parameters: an instance of `SemanticConfig` and an instance of `org.mozartspaces.core.config.Configuration`. The second one is allowed to be `null`. In that case, the default configuration will be used. A description of `SemanticConfig` can be found in the section 6.7. Depending on the triplestore and reasoner implementations used, different dependencies are necessary. The Maven projects and their dependencies are described in section 6.8.

Listing 7.1 shows the instantiation of a new Semantic MozartSpaces runtime. First, a semantic configuration is declared with: `SPARQLStore` as Semantic Store, followed by the Sesame triplestore, the OWLAPI as reasoner, the persistence strategy, and the directory "/home/user/sxvsm/" for storing persistent data. The default values are used for the remaining MozartSpaces settings.

```
SemanticConfig semanticConfig = new SemanticConfig()
  .setSemanticStoreClass(SparqlStore.class)
  .setSparqlEndpointClass(SesameEndpoint.class)
  .setContainerReasonerClass(OWLAPIContainerReasoner.class)
  .setPersistenceStrategy(PERSISTENCE_STRATEGIES.PERSISTENCE)
  .setPersistenceDirectory("/home/user/sxvsm/");
Capi capi = SemanticAPI.createSemanticCore(semanticConfig,
    CommonsXmlConfiguration.load(0));
```

Listing 7.1: Initialization of Semantic MozartSpaces

7.2 Container Creation

Only containers, including a semantic coordinator, are usable for semantic queries.

Listing 7.2 shows the creation of a new container on the core created in the previous section. This container has no restriction on the number of its entries. The Any coordinator is obligatory, and the semantic coordinator is optional. The name of the container and the transaction are allowed to be `null`.

```
ContainerReference container = capi.createContainer(
  containerName, null, MzsConstants.Container.UNBOUNDED,
  Arrays.asList(new AnyCoordinator()),
  Arrays.asList(new SemanticCoordinator()),
  transaction);
```

Listing 7.2: Creation of new container

In Semantic MozartSpaces, coordinators with own coordination data should have URIs as their names. Then, their coordination data, which are properties of entries, can be easily used in ontologies and semantic queries.

7.3 Writing Entries

Writing entries is done as with native MozartSpaces. If the value of an entry should be usable in semantic queries, then it has to be an instance of `SerializedSubResourceTree`. For less error-proneness and greater comfort, it is recommended to use as entry values special Java beans. Before creating an entry, such a Java bean should be converted by the object resource mapper to `SerializedSubResourceTree`. Classes of such Java beans must be prepared for the object resource mapper (see section 6.4). In the container-hosting MozartSpaces runtime, entry values that are Java literals are converted automatically to their XML Schema types. They do not need a special Java bean and mapping.

Listing 7.3 presents the creation of an entry and the insertion of this entry into a container. The class of the entry value and its instance are the same as in the listing 6.1. Listing 7.4 shows the resource tree that represents the written entry in the triplestore.

```
Set<URI> requiredSkills = new HashSet<>();
requiredSkills.add(URI.create("http://xvsm.org/tasks#cutting"));
requiredSkills.add(URI.create("http://xvsm.org/tasks#burning"));
Task task = new Task();
task.setName("Task 1");
task.setRequiredSkills(requiredSkills);
SerializedSubResourceTree entryValue = SemanticAPI
    .serializeEntryValue(task);
capi.write(container, new Entry(entryValue));
```

Listing 7.3: Writing semantic entry

```
@prefix t: <http://xvsm.org/tasks#> .
@prefix sxvsm: <http://xvsm.org/semantic#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
<http://xvsm.org/semantic/entry#1>
  rdf:type  sxvsm:Entry , sxvsm:ResourceTreeRoot ;
  <http://xvsm.org/semantic/coordinator/AnyCoordinator>
    sxvsm:Entry ;
  sxvsm:hasValue
  [ rdf:type t:Task ;
    t:hasName "Task 1"^^xsd:string ;
    t:requiresSkill t:burning , t:cutting
  ] ;
  sxvsm:id "1"^^xsd:long .
```

Listing 7.4: Written entry in triplestore

Metadata

In MozartSpaces, metadata of entries are commonly represented by coordination data. In same cases, metadata with no relation to a coordinator are required, when, for instance, an ontology requires special metadata for entries.

Such special metadata are added to an instance of `EntryMetadata` that is attached to the entry as coordination data. Such metadata must be the type of `SerializedSubResource-Tree`, and must be associated with a property URI.

7.4 Reading Entries

Semantic entries can, of course, be selected by any selector. This section describes the reading of entries with a semantic selector. Non-semantic entries can be also read with a semantic selector that uses their coordination data, or their literal value.

The code in listing 7.5 tries to read an entry. The value of that entry must be the type of `t:Task`, and requires at least the skill `t:cutting`. If the container contains only the entry that was written in the previous section, then this entry will be read. Listing 7.6 shows the SPARQL query that is generated by the semantic back-end to query the triplestore.

The function `createSimpleQuery` of `SemanticAPI` can be used for simple queries that use only the entry value. In the listing, it could be replaced by the following code:

```
SemanticSelector semanticSelector = SemanticAPI
    .createQuery("?entry <http://xvsm.org/semantic#hasValue> "
      + "[rdf:type t:Task ; t:requiresSkill t:cutting]");
```

```
SemanticSelector semanticSelector = SemanticAPI
    .createSimpleQuery("a t:Task ; t:requiresSkill t:cutting");
semanticSelector.getQuery()
    .addPrefix("t", URI.create("http://xvsm.org/tasks#"))
    .addPrefix("rdf", URI.create(
      "http://www.w3.org/1999/02/22-rdf-syntax-ns#"));
List<SerializedSubResourceTree> entryValueList = capi.read(
    container, semanticSelector, 1000, null);
Task readTask = SemanticAPI
    .deserializeEntryValue(entryValueList.get(0));
```

Listing 7.5: Reading semantic entry

To influence the order of the entries in a result of a read operation, ”ORDER BY” and ”GROUP BY” properties of the entry query (`EntryQuery`) have to be defined.

7.5 Ontology Management

Ontologies can be used to add knowledge to entries. Such attached data can be used by semantic queries or subscriptions. Each container has its own ontology that is the sum of all its ontology entries.

To extend the container ontology, a new ontology entry must be written. To remove a part of an ontology, the entry that includes it must be removed. An update on the ontology has only an impact on the entries that are written afterwards.

To easily find ontologies, especially for removing, the use of coordination data is recommended (e.g. a Key coordinator with `sxvsm:ontologyKey` as its name).

Ontology updates should be done carefully. If a container ontology is not consistent after an update, the container will reach an inconsistent state. The update operations will not have any

```
PREFIX t: <http://xvsm.org/tasks#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sxvsm: <http://xvsm.org/semantic#>
SELECT ?entryId
FROM <container:1>
FROM <container:1/inferred>
FROM NAMED <container:1/ontology>
WHERE {
  BIND (<container:1/ontology> AS ?ontology)
  ?entry sxvsm:id ?entryId ;
    rdf:type sxvsm:Entry .
  ?entry sxvsm:hasValue [
    rdf:type t:Task ;
    t:requiresSkill t:cutting
  ] .
}
GROUP BY ?entryId
```

Listing 7.6: SPARQL query generated by semantic coordinator

effect on the container ontology. But the ontology entries will be added (or removed) to (from) the container. The only feedback that will be given is a new log entry in the error log of the container hosting MozartSpaces runtime.

In listing 7.7, an OWL restriction is added to the container ontology. If this code is executed before the one shown in section 7.3, then the written entry value (”Task 1”) will become also an instance of the OWL class `t:CuttingTask`.

```
OntologyEntryValue ontologyValue = new OntologyEntryValue(
  "@prefix t: <http://xvsm.org/tasks#> . " +
  "@prefix owl: <http://www.w3.org/2002/07/owl#> . " +
  "@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>."+
  "t:CuttingTask rdf:type owl:Class ;\n" +
  " owl:equivalentClass [ rdf:type owl:Restriction ;\n" +
  "   owl:onProperty t:requiresSkill ;\n" +
  "   owl:someValuesFrom [ rdf:type owl:Class ;\n" +
  "     owl:oneOf ( t:cutting )\n" +
  "   ]\n" +
  " ].");
capi.write(container, new Entry(ontologyValue));
```

Listing 7.7: Adding ontology

Consistency Check

Ontologies can also be used for consistency checks. If an entry that is not consistent with the container ontology is inserted into the container, an exception `InconsistentEntryException` will be thrown, and the sub-transaction of the write operation will be rolled back.

In listing 7.8 the container ontology is extended with the OWL class `t:BadTask`. Then a new entry, including a task, is attempted to be written into the container. The written entry value, which is the type of `t:Task`, additionally becomes the type of `t:BadTask`. However, these two classes were declared as disjoint with each other. Hence, this write operation fails, because the entry is inconsistent with the container ontology.

```
ontologyValue = new OntologyEntryValue(
  "@prefix t: <http://xvsm.org/tasks#> ." +
  "@prefix owl: <http://www.w3.org/2002/07/owl#> . " +
  "@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>."+
  "t:BadTask rdf:type owl:Class ; " +
  " owl:equivalentClass [ rdf:type owl:Restriction ;" +
  "   owl:onProperty t:requiresSkill ;\n" +
  "   owl:someValuesFrom [ rdf:type owl:Class ; " +
  "       owl:oneOf ( t:spraying ) " +
  "   ] " +
  " ] ; " +
  " owl:disjointWith t:Task .");
capi.write(container, new Entry(ontologyValue));

requiredSkills = new HashSet<>();
requiredSkills.add(URI.create(
    "http://xvsm.org/tasks#spraying"));
Task badTask = new Task();
badTask.setName("Bad task");
badTask.setRequiredSkills(requiredSkills);
entryValue = SemanticAPI.serializeEntryValue(badTask);
// following operations throws InconsistentEntryException
capi.write(container, new Entry(entryValue));
```

Listing 7.8: Inconsistent entry

7.6 Semantic Notification

A user can subscribe for notifications about special operations. A semantic subscription includes an operation filter that contains an OWL class and a list of operation types (read, take, and write). For sending a notification, the value of the processed entry of an operation must be the type of the OWL class defined in the subscription.

The container storing notification entries must be on the same MozartSpaces core as the observed container. The observed container must include the notification emitter as an aspect. It can be registered with the following function:

```
SemanticAPI.registerNotificationEmitter(capi, container);
```

Listing 7.9 shows the creation and the insertion of a semantic subscription into the container that should be observed. This subscription is interested in read operations, including entry values of the type `t:CuttingTask`. The OWL class is included in the ontology in the listing 7.7.

During the execution of the listing 7.5, a notification will be created. An extra notification listener is not provided by Semantic MozartSpaces. This can be easily done by an extra thread that runs a blocking take operation, as the one in listing 7.10.

```
ContainerReference notificationContainer =
  capi.createContainer(null, null,
    MzsConstants.Container.UNBOUNDED,
    Arrays.asList(new LabelCoordinator(
      SemanticNotification.SUBSCRIPTION_ID_LABEL)),
    null, null);
SemanticSubscription subscription = new SemanticSubscription(
  URI.create("http://xvsm.org/tasks#CuttingTask"), "subId1",
  notificationContainer, Operation.READ);
capi.write(container,
  new Entry(SemanticAPI.serializeEntryValue(subscription)));
```

Listing 7.9: Semantic subscription

```
List<SerializedSubResourceTree> serNotifications = capi.take(
  notificationContainer, LabelCoordinator.newSelector("subId1",
    MzsConstants.Selecting.COUNT_ALL,
    SemanticNotification.SUBSCRIPTION_ID_LABEL), 10000,
  null);
SemanticNotification notification = SemanticAPI
  .deserializeEntryValue(serNotifications.get(0));
```

Listing 7.10: Reading notification

7.7 Recommendations

In a distributed application, the management of container ontologies and stored queries may have to be restricted. This can be done by the coordination-based access control. In the simplest form, operations, including entries with `OntologyEntryValue` as values, are restricted.

In Semantic MozartSpaces, coordination logic can be included in the following entities:

- (custom) coordinators
- (custom) aspects
- (stored) semantic queries
- ontology entries
- context entries

The advantage of coordination logic stored in ontologies is that it can be used by different queries, consistency checks, subscriptions, and other ontology parts. Additionally, ontologies can enhance entry values with implicit data that may be useful for a user that reads these entries. However, for coordination logic that dynamically changes, ontologies are not suitable. Changes in the ontology have an impact only on the newly written entries. Dynamically changing coordination logic should be formulated in semantic queries. This also concerns context dependent coordination logic.

CHAPTER 8

Concrete Use Case

This chapter presents a solution for a use case of task allocations in the area of multi agent systems.

The papers [DKKT13] and [DKKTon] present a similar problem and solutions, but without reasoning. Additionally, the second paper added positions to the tasks and the agents. The distances between an agent and the tasks influence the order of the task selection.

8.1 Problem

Several agents have to fulfill a mission together. A mission is divided into several tasks. Each task requires special, general skills (different specific skills are possible) for execution. Agents have different, specific skills. There is no connection between the tasks, and the order of their execution does not matter. The aim is a fast completion of a mission that can be achieved by ensuring a balanced load for all agents over the whole time. To simplify the use case all, the assumption was made that agents needed the same time for fulfilling every task.

The tasks and skills are an agricultural problem, where autonomous tractors (agents) have to work together on a field.

There are three different general skills with some specific skills:

- cutting
- flaming
  - flaming1
- spraying
  - spraying1
  - spraying2 (a different technique of spraying as in ”spraying1”)

Three different kinds of tasks that require different skills are produced:

1. flaming and cutting
2. cutting and spraying
3. spraying

There are three different task executor agents with the following skills:

1. cutting and flaming1
2. spraying1
3. cutting spraying2

8.2 Solution

All tasks are written into a container. Every time an agent has nothing to do, it will take a task out of the container.

A semantic query guarantees the correct task distribution. An agent gets only the tasks that it can execute. Therefore, the state of an agent, including its skills, is attached as a parameter to the semantic query. The strategy of task selection for balancing the work load is done as follows: An agent always gets the most difficult task that it can solve. The tasks which require more skills are more difficult. More highly skilled agents do not take simple tasks away from the less skilled agents.

The model for these tasks was presented in listing 6.1. Each task has a set of required skills. The state of an agent is modeled by the class in listing 8.1.

Listing 8.2 shows the ontology that is needed for modeling the hierarchy of the skills. That ontology can be split into several entries, for example, adding new skills during runtime. The property `t:subSkillOf` is declared explicitly. For each skill, the property `t:superSkillOf` is inferred by reasoning. This ontology will not enrich any semantic entry. Its data are used in the semantic query.

Listing 8.3 shows the semantic query and the Java code of an agent. A semantic selector with the agent's state as a context entry is build. A task is taken from the container. Then this task is executed. The result is written into the result container.

For each task, the semantic query calculates the number of required skills. Then the number of the overlapping skills of the task and of the agent are counted. If these two numbers are equal, then the agent will be able to execute the task, and the task will be selected. The tasks are sorted decrementally by the number of required skills.

The work balance of the use case implementation was tested with six agents (two instances for each agent type), and 90 tasks (30 instances for each type of task). Each agent was processed in its own thread. For simulating execution of tasks, the thread of an agent slept 300 ms. `SparqlStore` with `SesameEndpoint` was used as the Semantic Store. Each agent executed 15 tasks. The optimal balance of task execution was achieved.

```
@ResourceClass(defaultNamespace = "http://xvsm.org/tasks#")
public class TaskExecutorState implements Serializable {

  @ResourceProperty(name = "hasSkill")
  private Set<URI> skills;

  // placeholder for getters, setters,
  // and default constructor
}
```

resource class

```
[ rdf:type t:TaskExecutorState ;
  t:hasSkill t:cutting , t:flaming1 .
] .
```

agent of type 1

Listing 8.1: State of executor agents

In addition to the task distribution, the presented use case needs no extra network communication for coordination, or for exchanging the state of the agents. The performance of this use case application is evaluated in section 9.3.

8.3 Extension

The presented use case could be extended: A weight could be defined for the skills to consider their commonness in the calculation of the difficulty of a task. Tasks could require resources, and agents have resources. Tasks could have locations, and agents need some time to get there. The time an agent needs for task executions could also be of interest.

```
@prefix t: <http://xvsm.org/tasks#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

t:subSkillOf rdf:type owl:ObjectProperty ,
                      owl:ReflexiveProperty ,
                      owl:TransitiveProperty ;
             rdfs:range t:Skill ;
             rdfs:domain t:Skill ;
             owl:inverseOf t:superSkillOf .

t:superSkillOf rdf:type owl:ObjectProperty ;
               rdfs:domain t:Skill ;
               rdfs:range t:Skill .

t:Skill rdf:type owl:Class .

t:cutting rdf:type t:Skill, owl:NamedIndividual .

t:flaming rdf:type t:Skill, owl:NamedIndividual .

t:flaming1 rdf:type owl:NamedIndividual ;
           t:subSkillOf t:flaming .

t:spraying rdf:type t:Skill, owl:NamedIndividual .

t:spraying1 rdf:type owl:NamedIndividual ;
            t:subSkillOf t:spraying .

t:spraying2 rdf:type owl:NamedIndividual ;
            t:subSkillOf t:spraying .
```

Raw ontology

```
t:flaming t:subSkillOf t:flaming> ;
          t:superSkillOf t:flaming , t:flaming1 .
t:spraying t:subSkillOf t:spraying ;
           t:superSkillOf t:spraying, t:spraying1, t:spraying2 .
```

Inferred data

Listing 8.2: State of executor agents

```
?entry sxvsm:hasValue ?entryValue .
?entryValue rdf:type t:Task .
{
  select ?entryValue (count(?ns) as ?requiredSkills) {
    ?entryValue t:requiresSkill ?ns .
  } group by ?entryValue
}
{
  select ?entryValue (count(?ts) as ?overlappingSkills) {
    ?entryValue t:requiresSkill ?ts .
    GRAPH ?ontology { ?ts t:superSkillOf ?sts . }
    GRAPH ?context {
      ?contextEntry t:hasSkill ?sts .
    }
  } group by ?entryValue
}
FILTER (?requiredSkills = ?overlappingSkills)
```

WHERE part of semantic query

```
SemanticSelector semanticSelector = SemanticAPI.createQuery(
    wherePart, 1);
EntryQuery query = semanticSelector.getQuery();
query.addOrderByVariables("DESC(?requiredSkills)");

query.addPrefix("t", "http://xvsm.org/tasks#")
  .addPrefix("sxvsm", URI.create(Ontology.ONTOLOGY_URI))
  .addPrefix("rdf", URI.create(
        "http://www.w3.org/1999/02/22-rdf-syntax-ns#"));
semanticSelector.addContextEntry("context",
    SemanticAPI.serializeEntryValue(executorState));

tx = capi.createTransaction(
    MzsConstants.TransactionTimeout.INFINITE, null);
List<SerializedSubResourceTree> result = capi.take(
    newTaskContainer, semanticSelector, 1000, tx);

// simulation of task execution
Thread.sleep(300);
// writing result into result container
capi.write(resultContainer,
    MzsConstants.RequestTimeout.INFINITE, tx,
    new Entry(executionResult));

capi.commitTransaction(tx);
```

Java code

Listing 8.3: Executing tasks

CHAPTER 9

Evaluation

This chapter presents the differences between Semantic XVSM and the existing frameworks. It also contains a description of applications scenarios, and presents selected benchmarks of Semantic MozartSpaces, which demonstrates a practical feasibility of using Semantic XVSM.

9.1 Differentiation of Semantic XVSM

This section presents the advantages of, as well as the differences in Semantic XVSM, as compared to other semantic tuple spaces, XVSM, and a triplestore.

Difference from Semantic Tuples Spaces

In this subsection, Semantic XVSM, including its reference implementation Semantic MozartSpaces, is compared with the semantic tuple spaces projects presented in chapter 4.

The focus of the reviewed projects was on managing a distributed shared knowledge base. In contrast, the focus of my thesis lies on the communication between distributed agents.

A short summary of some vocabularies is warranted here: In XVSM, tuples are called entries, and a space is called a core, which is divided into several containers.

The following lists the differences, and the exclusive features of Semantic XVSM:

- Semantic XVSM introduced a new data model inspired by an X-tree and nested blank nodes. The original tuple spaces model also has a tree structure, but without labels for branches.

  The projects sTuples and TSC used an RDF graph as a tuple model. Every tuples is stored in an extra named graph. Some triplestores, like Jena, are not designed to execute SPARQL over a large number of named graphs. For large spaces, this could become a bottleneck. The data model for tuples of Semantic Tuple Centres is similar to the one of Semantic XVSM. But Semantic Tuple Centres support only a flat data structure without nested properties. In TripCom, a tuple is only an RDF triple. One triple cannot represent the same

information as an XVSM entry, cannot have a practical identifier, has excessive overhead for metadata, and does not permit different entries with the same value. The data model of TripCom is not compatible with SPARQL standard, and cannot be used with most of the existing triplestore implementations.

- The complete interaction of Semantic XVSM with a triplestore is defined with SPARQL. This enables a support for any SPARQL-compatible triplestore. Semantic queries will be able to use new features of future versions of SPARQL without big changes on Semantic XVSM.

- The possibility of Semantic XVSM to use different selectors in a chain, including non- and semantic selectors, is outstanding.

- XVSM is designed to support long-lived, concurrent transactions, and provides transactional timeouts with automatic rollback. These two features are also fully supported by Semantic XVSM, independent of the triplestore implementation used. None of the reviewed projects presented a concrete design for long-lived, concurrent transactions, or transactional timeouts.

- Semantic MozartSpaces can also store non-semantic entries (Java objects or literals) in containers. Even semantic queries can be used for such entries.

- In Semantic MozartSpaces, all entries are stored only in a triplestore. Only temporary weak-referenced Java objects of the entries exist additionally to the triplestore in the memory. Depending on the triplestore implementation, a Semantic MozartSpaces runtime can store more entries than can fit into the memory. In the implementations of the other projects, different representations of all entries exist concurrently in the memory.

- Semantic XVSM benefits from the last improvements of the new Semantic Web standards. This fact enabled more design options for a semantic tuple spaces concept. User applications also benefit from the more powerful query and reasoning functionalities. SPARQL features, like the federation service, which provides additional data for solving queries from other SPARQL endpoints, property paths and some other new features of SPARQL, were not available in the reviewed projects.

- Semantic XVSM uses a different strategy for reasoning than the reviewed projects. During entry insertion, inferred data are produced for the entry. These data are stored with its entry in the triplestore. With this solution, separate frameworks for the triplestore and the reasoner can be used. The use of the OWL-API with a broad reasoner implementation support became possible. Reasoning is done only for a small dataset—namely one entry. The number of entries in a container have no impact on the performance of reasoning. During the restart of a Semantic XVSM runtime, reasoning needs not be redone. This strategy enables optionally the exclusion of special entries from reasoning.

  The reviewed projects with reasoning support used the same framework for querying and reasoning. These frameworks for reasoning need to load all data of one space into the memory. The number of tuples in a space has an influence on the speed of reasoning.

- Semantic XVSM offers flexible ontology management by writing and removing entries. No API extension is necessary on client-side. The ontology can be easily read by reading the entries that include parts of the ontology. One drawback is that ontology updates have no influence on the already written entries. In future works, a procedure for updating the inferred data of existing entries can be implemented.

  In the reviewed projects with reasoning support, the ontology for a space has to be defined at space creation, and cannot be changed during runtime.

- In Semantic XVSM, a new, scalable subscription mechanism was introduced. Subscriptions are registered by writing entries. The filter criteria are formulated in the same ontology that can also be used by semantic queries. For each operation, to retrieve subscriptions only a simple read operation is necessary. In reviewed projects, queries are used as subscriptions. After each update of a space, all the queries have to be executed, and their result has to be checked. They also needed an API extension for subscriptions.

  In contrast to Semantic XVSM, TripCom offers also a subscription mechanism for observing the state of a space. In Semantic XVSM, this can be done by a replication of potentially relevant information, storing these data in an extra triplestore, and observing this triplestore. The replication can be done by semantic notification mechanism. This strategy enables an easy solution for observing the combined state of several distributed containers, and can use additional reasoning.

- Only, Semantic MozartSpaces offers an object resource mapper for mapping POJOs (plain old Java objects) to semantic entries.

- Semantic XVSM introduced context entries as a parameter for more flexible queries.

- In Semantic XVSM, semantic queries can be stored as entries (stored queries). Thus, a user does not need to deal with SPARQL. The complete coordination logic of a semantic tuples spaces application can be formulated with SPARQL and OWL, and can be stored in a space by using the standard API. That was not possible with the reviewed projects.

- For distributed joins, only TripCom offers a concept similar to the sub-queries of Semantic XVSM.

Difference from XVSM and MozartSpaces

This subsection presents the new possibilities and improvements of Semantic XVSM and Semantic MozartSpaces as compared to native XVSM and native MozartSpaces.

- In Semantic XVSM, the state of a runtime is represented in RDF graphs, with a similar data model as described by Craß [Cra10]. These graphs can be persisted. The persistence API described in this thesis has the same data structure as the meta model of the XVSM specification. Both also apply transactional locks on the same data level. The existing persistence layer for MozartSpaces does not have a general specification for XVSM.

- In Semantic XVSM, entries, beside coordinators and aspects, can also include coordination logic: ontology entries, stored queries, and subscription entries. The advantage of these new entities is that during run time they are easily manageable with the standard API. XVSM coordinators, on the other hand, can only be created during the instantiation of a container.

- The semantic coordinator offers new capabilities for entry selection. Table 9.1 presents features comparison of the XVSM query language and the semantic queries used in the semantic coordinator. The coordination-based access control model can have restrictions on the entry level. XVSM query allows guessing of the information about hidden entries. This is prevented in the design of semantic queries.

- With reasoning, implicit data can be inferred and attached to entries, independent from the knowledge of users.

- Before insertion, written entries can be validated against the ontology of a container (consistency check). Thus, a guarantee that all entries in a container conform to a special data model is enabled.

- The complete state of a MozartSpaces core is stored in a triplestore. With standardized tools this triplestore can be accessed for debugging. An API for such an access has to defined in future works.

- In native MozartSpaces, for each subscription a new aspect has to be created. The creation of an aspect is problematic over the network. An instance of an aspect is created on client-side, serialized, and sent to the server-side. The remote creation of aspects is a security risk.

  Semantic subscriptions are written as entries in the observed container. This operation can be easily restricted by the access control model. Furthermore, semantic subscription offers a powerful and fast filter mechanism. The notification feature of native MozartSpaces does not have any filter capability.

- In native MozartSpaces, the container-hosting Java runtime must load all Java classes used by any entry. In Semantic MozartSpaces, RDF graphs are exchanged. The Java runtime does not need to load any special classes depending on its hosting entries.

- Standardized domain models defined in OWL can be used in Semantic XVSM, and can increase interoperability.

Difference from Triplestores

Semantic XVSM uses a triplestore for storing and querying entries.

The primary application of triplestores are knowledge bases that can be accessed on the Web. The focus of Semantic XVSM lies in distributed coordination by means of sharing objects (entries). A knowledge base hosted by a triplestore can still be accessed by Semantic XVSM.

XVSM Query Language	*Semantic Query (SPARQL)*
sort*, reverse	order by (more powerful)
distinct (entry value)	distinct in a sub-select (not possible in current design, because of security restrictions)
element of	`IN` operator for set of values or `FILTER` operator for range and so on
both have the same relational operators	
universal quantifier ”for all”	`NOT EXISTS` with the inverted function
logical ”and”	all statements of a query template are connected by default with an logical ”and”
logical ”or”	`OPTIONAL`s in combination with `FILTER` to check their bounds
logical negation	`NOT EXIST`, or logical negation of `FILTER`
path	property path (has more functionality)
-	sub-select
-	use of inferred data
-	distributed join (sub-queries and federation)
-	context entries as query parameters
-	W3C standard

Table 9.1: Feature comparison of XVSM query language and Semantic XVSM queries

The following list presents properties of Semantic XVSM that are not provided by most of the plain triplestores:

- management of resource trees,
- long-lived, concurrent transactions,
- transactional timeout with automatic rollback,
- coordinators (e.g. Fifo) and their combination in a chain,
- aspects (interceptors),
- access control on the resource tree level,
- subscription mechanisms on the resource tree level, and
- stored queries.

9.2 Applications Scenarios

Semantic XVSM is a general, low-level framework for dealing with complex coordination problems. Primarily, it can be used as a foundation for other more specialized frameworks, like an enterprise service bus, or a semantic workflow framework. The following features could be extended by the frameworks that use Semantic XVSM:

- *Distributed reasoning* can be implemented by updating the dataset of a central reasoner by using semantic notifications. For load balancing, the reasoning tasks can be divided into smaller sub-tasks that can be executed by several software agents.

- A *space state subscription* mechanism could notify as soon as a user-defined state over several spaces is reached. This could be implemented by a separate triplestore that has a snapshot of the relevant informations of the spaces. This triplestore could be updated with semantic notifications. It could be observed as described in Murth [Mur10].

- A *mediation* mechanism for applications with different data models could be implemented with one of the following strategies: For few different models with small differences, container ontologies could include mapping rules. The data of a read entry would be a combination of all data models. Another strategy would be the implementation of a mapper in an aspect, and the decision for one core data model that is stored in the triplestore of Semantic XVSM. The mapping of complex SPARQL queries would, however, be a significant challenge.

- For decentralized and distributed networks, Semantic XVSM could be combined with a peer-to-peer framework. The resulting framework could include automatic container discovery, different replication strategies, and many more.

Application scenarios with a flexible, dynamic environment and a variable number of heterogeneous software agents can benefit the most from the features of Semantic XVSM. For example, ubiquitous computing applications have such properties. Such applications have a different behavior depending on the current context. Building automation is a sub-part of ubiquitous computing.

The master worker pattern is convenient to be implemented with tuples spaces as well as with Semantic XVSM. This pattern can be used to solve the problem of task allocations. The use case application of this thesis and the two scientific papers [DKKT13] and [DKKTon] presented a multi-agent system use case that was solved with Semantic MozartSpaces. Several agriculture robots have to solve a mission together. Such a mission is split into tasks that require special skills, resources for execution and are linked to an area. A robot that has nothing to do asks the semantic space for its next task. The presented use case needs no extra communication for coordination, or exchanging the state of the agents. The task selection is optimized for a balanced work load over all agents.

Common master worker applications also have tasks that requires skills, and resources for executions and have context-dependent conditions.

Applications in domains with existing OWL ontologies (e.g. some medical domains) can use the already defined data models in Semantic XVSM. Thus, communication and coordination can benefit from the implicit data and the public available domain knowledge.

Even applications on a single computer can use Semantic XVSM as an object store with SPARQL selection, or for modularization, where software components coordinate with each other over a local semantic space.

9.3 Benchmarks

The benchmarks presented in this section tested the performance of Semantic MozartSpaces. The time needed to read and write 1,000 entries was measured. Querying and writing tasks of the presented use case was also tested. The results of these benchmarks should serve as an indicator of the practical feasibility of Semantic MozartSpaces.

All benchmarks were tested on a notebook with the following properties:

- operation system: Ubuntu™13.10 64-bit
- Java™ version: OpenJDK™ 1.7.0_25
- processor: Intel® Core™ i5-3230M CPU (2.60GHz x 4)
- memory: 12 GiB
- disk: SSD Samsung® 840 Pro
- Eclipse™ 4.3 with the settings ”-Xms256m” and ”-Xmx512m”

All Java classes that were used for the benchmarks were stored in the Maven project ”semanticspaces-test”.

Mass Read and Write

In this subsection, the benchmarks tested 1,000 read and write operations using non-semantic entries composed of a simple string, and an integer. An Any coordinator and a Key coordinator managed all entries. The test used one single, local container without a reasoner. All operations were processed sequentially inside a single thread.

Each test was executed with the following Store API implementations:

- native MozartSpaces (persistent profile ”BerklyDB transactional” for persistency tests)
- the Semantic Store `JenaStore`
- the Semantic Store `SparqlStore` with `JenaEndpoint`
- the Semantic Store `SparqlStore` with `SesameEndpoint`

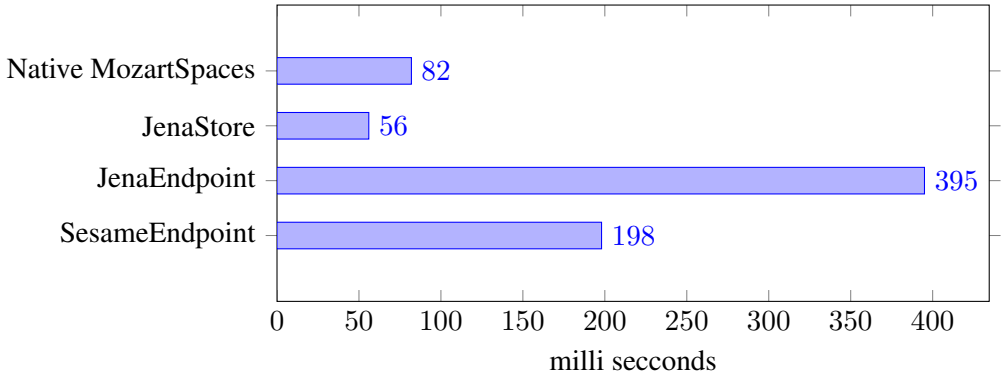

Figure 9.1: Sequential read of 1,000 persisted entries

The caching functionality of these implementations was not used.

To prevent the Java runtime optimizer influencing the results, each test was executed twice before measuring. Then each test ran ten times. The results were sorted, and the mean of the six middle test results was shown in the charts.

For testing the read operations, the Key coordinator was used to select an entry. Because the Key coordinator implementation has its own internal index structure, it made no difference which Store API was used, as long as all entries were present in the memory. Therefore, only persistent read tests were done. 1,000 entries were stored into a container. Then to ensure that no entries were present in the memory, the MozartSpaces runtime was restarted. Next, the time for reading the stored 1,000 entries was measured. Figure 9.1 presents the results for each Store API implementation. The implementation of the Store API with native Jena was more than 10 times faster than the general implementation with SPARQL. The code of the read benchmark implementation can be found in the class `org.mozartspaces.semantic.benchmarks.EntriesReadBenchmark`.

The chart in figure 9.2 lists the times that the different Store API implementations needed to process in-memory 1,000 write operations. The container test was empty at the start. Figure 9.3 presents the results for the same test with persistency. The results showed that the time that the triplestores needed for persisting, excludes Semantic MozartSpaces for many application scenarios at the moment. Jena needed more than 200 megabytes of disk space for the 1000 entries. In comparison, Sesame used less than one megabyte. The code of the write benchmark implementation can be found in the class `org.mozartspaces.semantic.benchmarks.EntriesWriteBenchmark`.

To ensure practical feasibility of Semantic MozartSpaces, the write performance has to be improved by future works. An optimized implementation of the Semantic Store with Sesame would be the best choice for most application scenarios.

In contrast to native XVSM, the use of Semantic Web technologies creates an overhead. At the moment, this fact restricts its use to applications that do not have to process a lot of concurrent write operations. Because Semantic XVSM enables more complex operations than native XVSM, less operations are perhaps necessary to fulfill the same tasks.

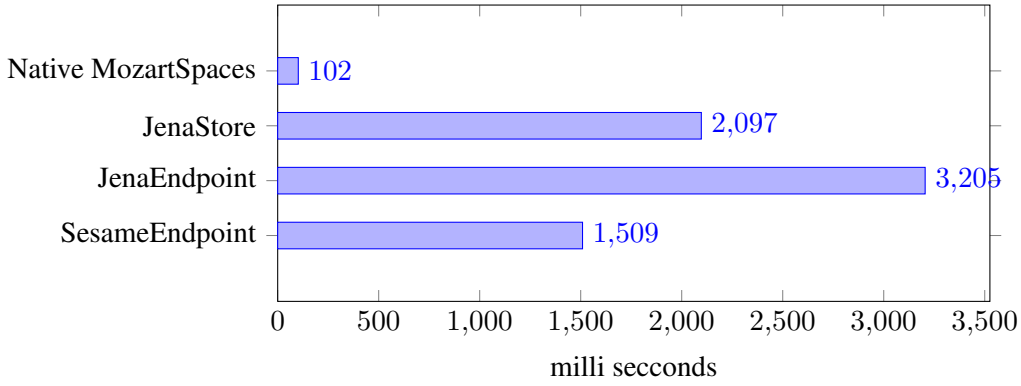

Figure 9.2: Sequential write of 1,000 entries (in-memory)

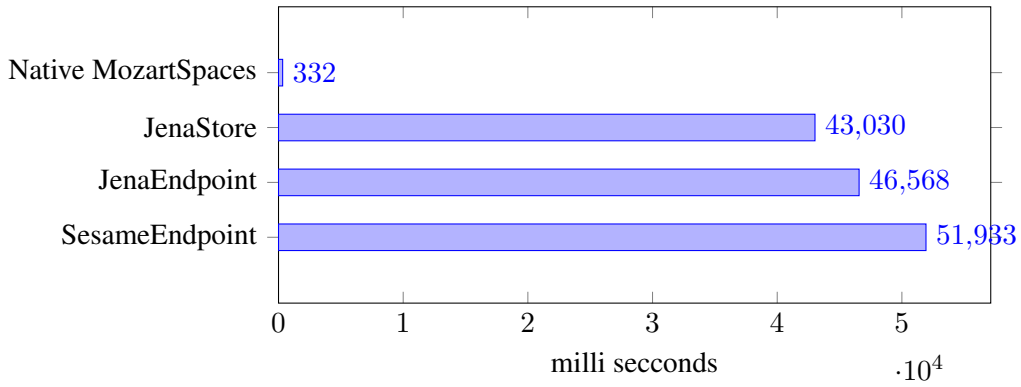

Figure 9.3: Sequential write of 1,000 entries (persistent)

Use Case

Additional tests were made with the use case implementation described in chapter 8. In this subsection, the time needed for querying, writing and removing of task entries is shown. All tests were done in-memory and with local Semantic MozartSpaces runtime.

The execution of one task by an agent is done as follows:

1. start a transaction,
2. executed a take operation to retrieve a task from the task container,
3. simulate the execution of the read task: the agent's thread sleeps for 300 ms,
4. write the execution result into the result container, and
5. commit the transaction.

There were three execution agents (one of each type), each running in their own threads.

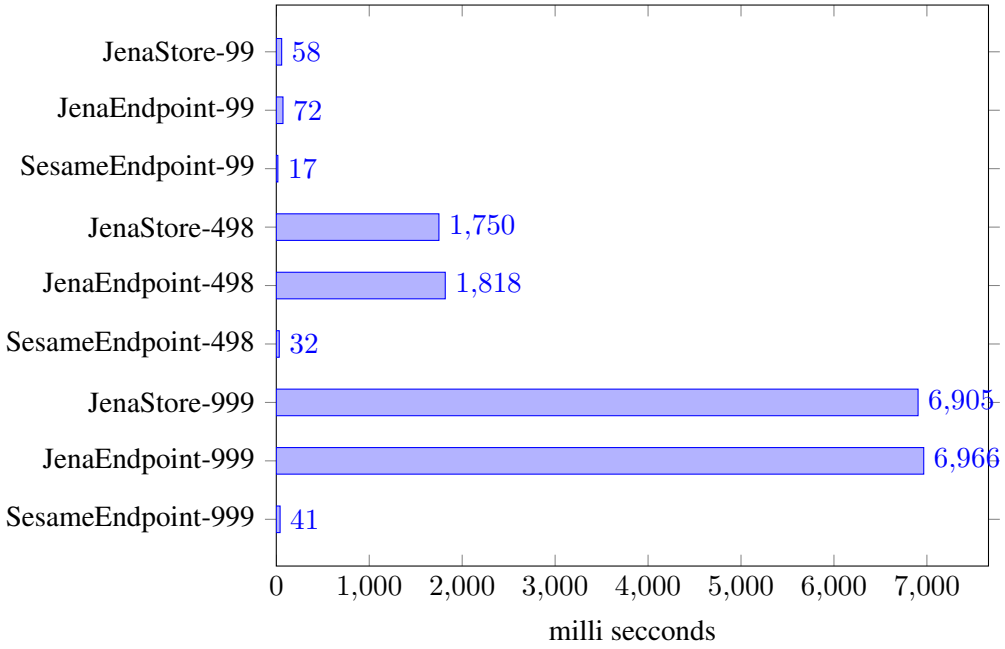

Figure 9.4: Average time required for retrieving one task entry

The benchmarks tested the processing of the first 30 tasks of a container that included 99, 498, and 999 tasks. The number of each task type was balanced (e.g. 99 tasks: 33 times type 1, 33 times type 2, and 33 times type 3). An extra thread executed a blocking read for 30 entries on the result container. Once this read operation could be performed the complete execution was stopped.

The following measurements were done from the perspective of the executor agents: Figure 9.4 shows the average time that was needed to read one task entry. The number of entries in a container influenced dramatically the query time of the Jena implementations. The Sesame implementation, even with 1000 entries, was very fast. Figure 9.5 shows the average time that was needed to write one entry into the result container. Figure 9.6 lists the times for committing the take and the write operations of one task execution. During a commit the entry is removed from the task container, and the write lock is removed from the entry in the result container.

For the use case presented in this thesis the Sesame implementation was clearly the best performing Semantic Store. An implementation with the native API of Sesame would perform even better. The Semantic Store implementations using Jena scales poorly.

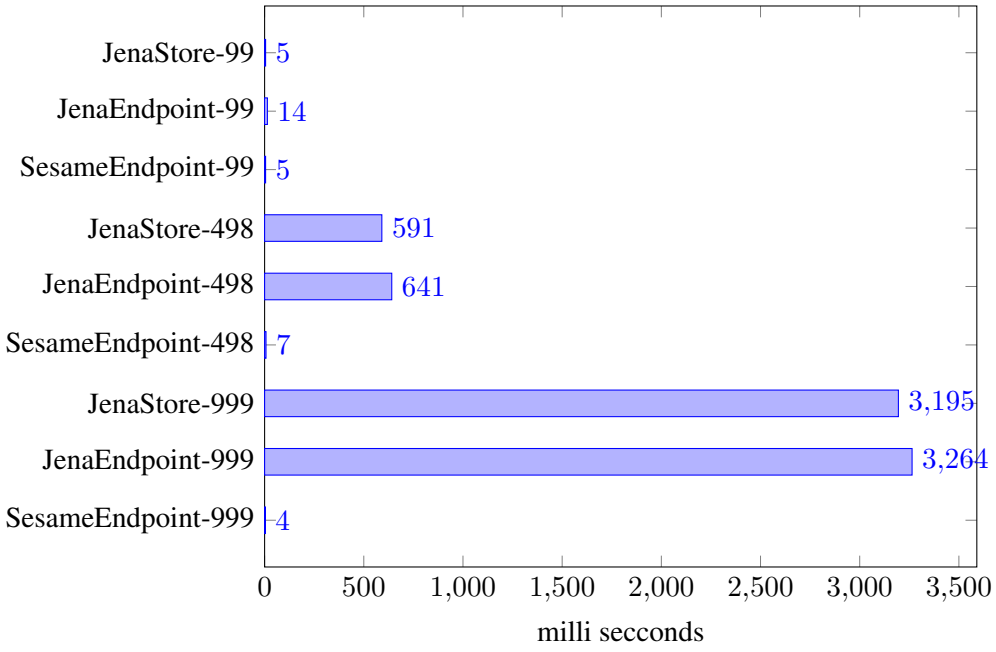

Figure 9.5: Average time required for writing one result entry

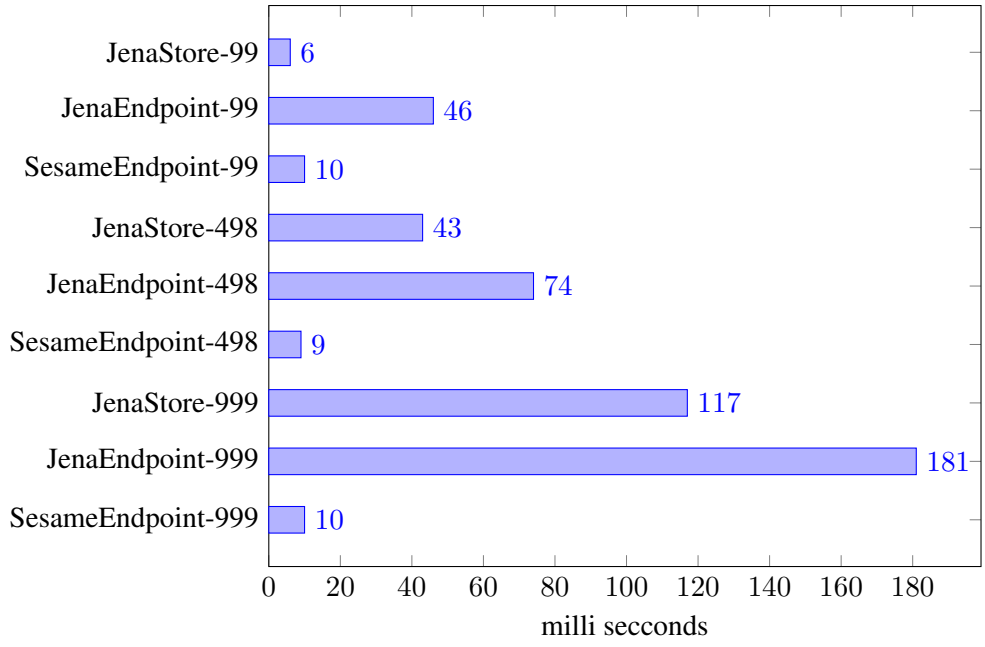

Figure 9.6: Average time required for committing one task execution

CHAPTER 10

Future Work

There is a considerable potential of enriching XVSM with semantic technologies. This chapter presents the ideas that were not implemented in this thesis, but could be of interest to future endeavors.

- Semantic XVSM could be the foundation for frameworks with a more domain specific focus.
- SPIN (SPARQL Inferencing Notation) [KHI11] could be evaluated for reasoning support with dynamically changing ontologies, and for improving stored queries. It is perhaps possible that with this technology a mechanism could be developed for supporting context dependent ontologies with different reasoning results that would depend on the current context.
- For pointing out the potential of Semantic XVSM, more use case scenarios and representative patterns could be described.
- The query predecessor could be extended to enable semantic queries that use other entries in a container as context information, without additional user-specified sub-queries. This extension would automatically extract these query parts into extra sub-queries.
- In addition to XML, the REST interface of MozartSpaces could be extended to support a Turtle syntax.
- A semantic look-up service for containers that uses a container description ontology extending OWL-S could be done. An extra container could be used as the registry for all container descriptors.
- For security reasons a restriction for accessing coordination data and inferred data should be of interest.

- The ontology management could be extended to support the update of the inferred data of the existing entries. However, such an extension would violate the rule that written entries are not modifiable.
- A mechanism for dependency management could prevent the remove of ontology entries that are still needed by other entries (semantic, query, ontology, or subscription entries).
- For systems that need extra long-lived transactions, a recovery mechanism for transactions after a system crash or a reboot could be implemented.
- There are already plans for extending MozartSpaces with leases for entries. Once implemented, entries will have a specified lifetime, and will be automatically removed after their expiry.

CHAPTER 11

Summary

This thesis introduced a new framework for supporting the development of distributed applications in a heterogeneous, dynamically-changing environment. Semantic Web technologies are suited for data exchange in distributed systems. They are designed to enable the creation of domain models from different parties over the Internet. Semantic MozartSpaces can be used to test how different application scenarios can benefit from semantic coordination. The use case implementation presented the power of formulating the coordination logic with Semantic Web technologies.

In this thesis, XVSM was extended, among others, with:

- a semantic selector,
- a reasoning mechanism with dynamic ontology management,
- a consistency check of entries against an ontology, and
- a semantic notification mechanism.

Semantic Web technologies develop rapidly. With Semantic XVSM, these technologies can be used for coordinating distributed agents. Well tested and publicly accepted domain models written in OWL can simplify the communication in heterogeneous systems. Knowledge published via SPARQL endpoints can support the coordination.

The results of this thesis can be primarily perceived as an input for new research projects. To better prove practical feasibility of Semantic XVSM, larger use case applications than the one presented in this thesis need to be developed and tested.

Bibliography

[Bar10] Martin-Stefan Barisits. Design and implementation of the next generation XVSM framework. Master's thesis, Vienna University of Technology, 2010. http://permalink.obvsg.at/AC07808083.

[BBLPC13] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. Turtle. W3C candidate recommendation, February 2013. http://www.w3.org/TR/2013/CR-turtle-20130219/.

[BFH+12] Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, and Michael Smith. OWL 2 web ontology language: Structural specification and functional-style syntax (second edition). W3C recommendation, December 2012. http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/.

[BG04] Dan Brickley and R.V. Guha. RDF vocabulary description language 1.0: RDF schema. W3C recommendation, February 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[BLFM98] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform Resource Identifiers (URI): Generic Syntax. W3C recommendation, August 1998. http://www.isi.edu/in-notes/rfc2396.txt.

[Brü13] Andreas Brückl. Relaxed non-blocking distributed transactions for the eXtensible virtual shared memory. Master's thesis, Vienna University of Technology, 2013. http://permalink.obvsg.at/AC10774537.

[CCG+12] Diego Calvanese, Jeremy Carroll, Giuseppe De Giacomo, Jim Hendler, Ivan Herman, Bijan Parsia, Peter F. Patel-Schneider, Alan Ruttenberg, Uli Sattler, and Michael Schneider. OWL 2 web ontology language: Profiles (second edition). W3C recommendation, December 2012. http://www.w3.org/TR/2012/REC-owl2-profiles-20121211/.

[CDJ+13] Stefan Craß, Tobias Dönz, Gerson Joskowicz, eva Kühn, and Alexander Marek. Securing a space-based service architecture with coordination-driven access control. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 4:76–97, March 2013. http://isyou.info/jowua/papers/jowua-v4n1-4.pdf.

[CG89] Nicholas Carriero and David Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Comput. Surv.*, 21(3):323–357, September 1989. http://doi.acm.org/10.1145/72551.72553.

[CKS09] Stefan Craß, eva Kühn, and Gernot Salzer. Algebraic foundation of a data model for an extensible space-based collaboration protocol. In *IDEAS*, pages 301–306, 2009. http://dx.doi.org/10.1145/1620432.1620466.

[Cra10] Stefan Craß. A formal model of the Extensible Virtual Shared Memory (XVSM) and its implementation in Haskell : design and specification. Master's thesis, Vienna University of Technology, 2010. http://permalink.obvsg.at/AC07806750.

[DKKT13] Domagoj Drenjanac, Lukas Klausner, eva Kühn, and Slobodanka Dana Kathrin Tomic. Semantic shared spaces for task allocation in a robotic fleet for precision agriculture. In Emmanouel Garoufallou and Jane Greenberg, editors, *Metadata and Semantics Research*, volume 390 of *Communications in Computer and Information Science*, pages 440–446. Springer International Publishing, 2013. http://dx.doi.org/10.1007/978-3-319-03437-9_43.

[DKKTon] Domagoj Drenjanac, Lukas Klausner, eva Kühn, and Slobodanka Dana Kathrin Tomic. Harnessing coherence of area decomposition and semantic shared spaces for task allocation in a robotic fleet. submitted for publication.

[Dön11] Tobias Dönz. Design and implementation of the next generation XVSM framework: runtime, protocol and API. Master's thesis, Vienna University of Technology, 2011. http://permalink.obvsg.at/AC07810664.

[FHA99] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces: Principles, Patterns and Practices*. Addison-Wesley Professional, 1999.

[FKS+07] Dieter Fensel, Reto Krummenacher, Omair Shafiq, eva Kühn, Johannes Riemer, Ying Ding, and Bernd Draxler. Tsc: Triple space computing. *e & i Elektrotechnik und Informationstechnik*, 124(1-2):31–38, 2007. http://dx.doi.org/10.1007/s00502-006-0408-1.

[GPP13] Paul Gearon, Alexandre Passant, and Axel Polleres. SPARQL 1.1 update. W3C recommendation, March 2013. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[HD05] Andreas Harth and Stefan Decker. Optimized index structures for querying rdf from the web. In *Web Congress, 2005. LA-WEB 2005. Third Latin American*, pages 10–pp. IEEE, 2005. http://sw.deri.org/2004/06/yars/.

[Hir13] Jürgen Hirsch. An adaptive and flexible replication mechanism for mozartspaces, the xvsm reference implementation. Master's thesis, Vienna University of Technology, 2013. http://permalink.obvsg.at/AC07814166.

[HS13] Steve Harris and Andy Seaborne. SPARQL 1.1 query language. W3C recommendation, March 2013. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[KHI11] Holger Knublauch, James A. Hendler, and Kingsley Idehen. SPIN - Overview and Motivation. W3C member submission, February 2011. http://www.w3.org/Submission/spin-overview/.

[KLF04] D. Khushraj, O. Lassila, and T. Finin. sTuples: semantic tuple spaces. In *Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004. The First Annual International Conference on*, page 268?277, 2004. http://dx.doi.org/10.1109/MOBIQ.2004.1331733.

[KSdf+08] Reto Krummenacher, Elena Simperl, doug foxvog, Vassil Momtchev, Dario Cerizza, Lyndon Nixon, Davide Cerri, Brahmananda Sapkota, Kia Teymourian, Philipp Obermeier, Daniel Martin, Hans Moritsch, Omair Shafiq, and David de Francisco. TripCom deliverable 6.5: Towards a scalable triple space. April 2008. http://tripcom.org/docs/del/D6.5.pdf.

[KSM+07] Reto Krummenacher, Elena Simperl, Vassil Momtchev, Lyndon Nixon, and Omair Shafiq. TripCom Deliverable 2.2: Specification of the Triple Space Ontology. March 2007. http://www.tripcom.org/docs/del/D2.2.pdf.

[Küh94] eva Kühn. Fault-tolerance for communicating multidatabase transactions. In *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*, volume 2, pages 323–332. IEEE, 1994. http://dx.doi.org/10.1109/HICSS.1994.323251.

[LSC+09] Michael Lafite, Christian Schreiber, Francesco Corcoglioniti, Alessio Carenini, Philipp Obermeier, and Sebastian Dill. TripCom deliverable 6.4: Component integration, demonstration, and documentation. March 2009. http://www.tripcom.org/docs/del/D6.4.pdf.

[MB04] Ashok Malhotra and Paul V. Biron. XML schema part 2: Datatypes second edition. W3C recommendation, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[MJK+07] Martin Murth, Gerson Joskowicz, eva Kühn, Dario Cerizza, Davide Cerri, David de Francisco, Alessandro Ghioni, Reto Krummenacher, Daniel Martin, Lyndon Nixon, Nuria Sanchez, Brahmananda Sapkota, Omair Shafiq, and Daniel Wutke. TripCom deliverable 6.2: Triple space reference architecture. April 2007. http://www.tripcom.org/docs/del/D6.2.pdf.

[MM04] Eric Miller and Frank Manola. RDF primer. W3C recommendation, February 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[Mur10] Martin Murth. *Efficient coordination with semantic shared data spaces*. PhD thesis, Vienna University of Technology, 2010. http://permalink.obvsg.at/AC07807221.

[Nar11] Elena Nardini. *Semantic coordination through programmable Tuple spaces*. PhD thesis, University of Bologna, April 2011. http://amsdottorato.cib.unibo.it/3477/.

[NOV13] Elena Nardini, Andrea Omicini, and Mirko Viroli. Semantic tuple centres. *Science of Computer Programming*, 78(5):569–582, May 2013. http://dx.doi.org/10.1016/j.scico.2012.10.004.

[NSKMR08] Lyndon Nixon, Elena Simperl, Reto Krummenacher, and Francisco Martin-Recuerda. Tuplespace-based computing for the semantic web: a survey of the state-of-the-art. *The Knowledge Engineering Review*, 23(02), May 2008. http://dx.doi.org/10.1017/S0269888907001221.

[Omi06] Andrea Omicini. Formal ReSpecT in the A&A perspective. In Carlos Canal and Mirko Viroli, editors, *5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06)*, pages 93–115, CONCUR 2006, Bonn, Germany, August 2006. University of Málaga, Spain. Proceedings.

[OZ99] Andrea Omicini and Franco Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, 1999. http://dx.doi.org/10.1023/A%3A1010060322135.

[RKL+05] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Cristoph Bussler, and Dieter Fensel. Web service modeling ontology. *Appl. Ontol.*, 1(1):77–106, January 2005. http://dl.acm.org/citation.cfm?id=1412350.1412357.

[SMS08] Brahmananda Sapkota, Vassil Momtchev, and Omair Shafiq. TripCom deliverable 1.3: High-performance storage implementation. March 2008. http://www.tripcom.org/docs/del/D1.3.pdf.

[Zar12] Jan Zarnikov. Energy-efficient persistence for extensible virtual shared memory on the android operating system. Master's thesis, Vienna University of Technology, 2012. http://permalink.obvsg.at/AC07813273.

Web References

[1] The BSD 3-Clause License. http://opensource.org/licenses/BSD-3-Clause, 1998. University of California, Berkeley.

[2] Apache License, version 2.0. http://www.apache.org/licenses/LICENSE-2.0, January 2004. The Apache Software Foundation.

[3] AGPL - GNU Affero General Public License, version 3. http://www.gnu.org/licenses/agpl-3.0.html, November 2007. Free Software Foundation, Inc.

[4] TSC - project website. http://tsc.deri.at/, 2007.

[5] ORDI - project website. http://ordi.sourceforge.net/, 2009.

[6] TripCom - project website. http://tripcom.org/, 2009.

[7] tsc++ - project website. http://tsc.sti2.at/, 2009.

[8] Apache Jena. https://jena.apache.org/, 2013.

[9] DBpedia. http://dbpedia.org/, 2013.

[10] ELK reasoner - project website. http://code.google.com/p/elk-reasoner/, 2013.

[11] FaCT++ reasoner - project website. http://code.google.com/p/factplusplus/, 2013.

[12] HermiT reasoner - project website. http://hermit-reasoner.com/, 2013.

[13] JFact DL Reasoner - project website. http://jfact.sourceforge.net/, 2013.

[14] Mozartspaces. http://www.mozartspaces.org/, 2013.

[15] Owlim - product website. http://www.ontotext.com/owlim, 2013.

[16] Pellet reasoner - project website. http://clarkparsia.com/pellet, 2013.

[17] Protégé. http://protege.stanford.edu/, 2013.

[18] Sesame. http://www.openrdf.org/, 2013.

[19] The OWL API - project website. http://owlapi.sourceforge.net/, 2013.