Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT FÜR !NFORMATIK

A Formal Model of the Extensible Virtual Shared Memory (XVSM) and its Implementation in Haskell

Design and Specification

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Stefan Craß
Matrikelnummer 0325656

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuerin: A.o. Univ. Prof. Dr. Dipl.-Ing. eva Kühn
Mitbetreuer: A.o. Univ. Prof. Dr. Dipl.-Ing. Gernot Salzer

Wien, 05.02.2010

(Unterschrift Verfasser) (Unterschrift Betreuerin)

Stefan Craß, Bründlfeldweg 63/1/6, 7000 Eisenstadt

„Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel 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."

Wien, 05.02.2010 _____

(Unterschrift)

Abstract

The development of distributed applications is a complex task that requires efficient communication and coordination between all participants. The space-based computing paradigm (SBC) enables simple collaboration between different peers due to a data-driven interaction style. This thesis describes the formal specification of XVSM (eXtensible Virtual Shared Memory), which represents a flexible and extensible SBC middleware that allows loosely coupled systems to coordinate themselves efficiently. Based on a simple algebraic foundation and an expressive query language, the semantics of the middleware's core functionality are defined via the specification of modules for basic data access, transactions, coordination and the runtime machine. A meta model is defined for XVSM to bootstrap the behavior of the space with own mechanisms. It is also shown how the middleware can be adapted to support arbitrary coordination laws that exceed the default semantics. The XVSM specification has been used to implement an executable prototype with the functional programming language Haskell. The feasibility of the formal model is proven with this XVSM prototype, for which the architecture and implementation are described in this thesis.

Kurzfassung

Die Entwicklung verteilter Applikationen ist eine komplexe Aufgabe, die eine effiziente Kommunikation und Kollaboration aller Beteiligten erfordert. Das Space-Based-Computing-Paradigma (SBC) ermöglicht die einfache Zusammenarbeit verschiedener Peers durch einen datengetriebenen Interaktionsstil. Diese Diplomarbeit beschreibt die formale Spezifikation von XVSM (eXtensible Virtual Shared Memory), einer flexiblen und erweiterbaren SBC-Middleware, die die effiziente Koordination von lose gekoppelten Systemen ermöglicht. Ausgehend von einer einfachen algebraischen Basis und einer ausdrucksstarken Abfragesprache wird die Semantik der Grundfunktionalität der Middleware beschrieben. Dies geschieht durch die Spezifikation der Module für grundlegenden Datenzugriff, Transaktionen, Koordination und die Runtime-Maschine. Für XVSM ist ein Meta-Modell definiert, das es ermöglicht, das Verhalten des Spaces mit eigenen Mechanismen zu erklären. Zusätzlich wird beschrieben, wie beliebige Koordinationsformen, die über das Standardverhalten hinausgehen, in die Middleware integriert werden können. Die XVSM-Spezifikation wurde benutzt um einen lauffähigen Prototyp in der funktionalen Programmiersprache Haskell zu implementieren. Durch die Entwicklung dieses XVSM-Prototyps, dessen Architektur und Implementierung in dieser Arbeit beschrieben wird, wird die Realisierbarkeit des formalen Modells gezeigt.

The specification of XVSM described in this thesis would not have been possible without the help of several people that have worked or currently work on the concepts of the XVSM technology.

I would like to thank my supervisor, *eva Kühn*, and *Gernot Salzer*, who have developed the basics of the formal model with me in numerous meetings and helped me to resolve several open issues.

I also want to thank the members of the Space Based Computing Group and the XVSM Technical Board, who have given valuable feedback to my work, especially *Richard Mordinyi*, *Laszlo Keszthelyi*, *Christian Schreiber*, *Martin Barisits*, *Alexander Marek* and *Tobias Dönz*.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

2PL	Two-phase locking
API	Application programming interface
CAPI	Core application programming interface
CoXT	Coordination Xtree
CSP	Communicating Sequential Processes
EBNF	Extended Backus-Naur Form
FIFO	First-in-first-out
GHC	Glasgow Haskell Compiler
JMS	Java Messaging Service
LIFO	Last-in-first-out
P2P	Peer-to-peer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SBC	Space-Based Computing
SOS	Structural Operational Semantics
SQL	Structured Query Language
STM	Software Transactional Memory
SXQ	Simple XVSM Query
TLA	Temporal Logic of Actions
TP	Timeout processor
URI	Uniform Resource Identifier
XML	Extensible Markup Language
XP	XVSM core processor
XQ	XVSM Query
XVSM	Extensible Virtual Shared Memory
XVSMP	Extensible Virtual Shared Memory Protocol
XVSMQL	Extensible Virtual Shared Memory Query Language

1 Introduction

The internet enables its users to acquire information and communicate with people all over the world. Using this infrastructure, sophisticated software applications can be built that allow various software processes on distributed peers to collaborate with each other in a highly dynamic manner. The coordination of these processes is a complex task that can be mastered by an intelligent utilization of available resources and efficient communication among peers, together with an easily adaptable programming model. Classical client-server architectures, where one central host carries out most computations, usually do not offer the necessary flexibility and scalability to compete in dynamic scenarios because the limited computing capacity and network bandwidth of the server create a bottleneck for the whole application. Therefore, the peer-to-peer paradigm (P2P) [1] was invented that allows every peer to act as a client and as a server at the same time. Peers can be seen as nodes of a network that allows ad-hoc connections between arbitrary users, which enables direct collaboration between peers that do not need to know each other beforehand. Additionally, complex tasks can be split up among several peers to make efficient use of available computing power.

The coordination of multiple peers is a sophisticated task. Problems like concurrent access, synchronization, heterogeneous systems and the cognition of network failures should not be solved by application programmers. Instead, middleware systems are used that hide low-level communication mechanisms, allowing the programmer to focus on the content of the sent data as well as the business logic without having to think about how the communication is established. With the help of an appropriate middleware, easily adaptable distributed applications can be created in an efficient way. Beside remote communication and data access, a middleware may also support further functionality that simplifies the programming effort, e.g. transactions, automatic replication, persistency, recovery after crashes and dynamic lookup of peers via names or certain properties.

Middleware systems like RPC [46] and RMI [49] are based on the ability to call a method on a remote computer similar to a local method invocation. Message-oriented middleware adds an additional layer between two communicating peers in the form of message queues. Both peers can send and receive messages that may contain data or requests. The messages are buffered in the message queue, so temporal decoupling is achieved as both peers need not be online at the same time for a successful interaction. However, both middleware approaches have in common that they favor the client-server paradigm, where the clients need to register to a well-known server that executes their requests. When using one of these middleware systems, P2P applications would need a non-scalable amount of bidirectional connections between every peer, which would have to be established and synchronized by the programmer. If additional peers are added

to fulfill a task, a high programming overhead would be needed to balance the load of incoming requests appropriately. A feasible solution to these problems is a paradigm change from message-based communication to data-driven applications. Instead of sending requests to known peers, they are written to a common blackboard, a so-called virtual shared memory or space. The participating processes, which may run on the same or on different machines, do not need to know each other and may wait for particular requests, responses and other data, which can be provided by other peers via the space. Thus, an application only needs to communicate with a single space, similar to a simple client-server architecture. However, the data-driven approach enables the asynchronous coordination of autonomous peers without any central server application that must be aware of all clients, thus enabling the temporal, referential and spatial decoupling of peers. Therefore, peers of a distributed application can dynamically join and leave without any problem, and they can coordinate themselves in a P2P style without any direct communication. The space itself may run on a single machine but it could also be distributed on several peers to gain the performance advantages of P2P. This behavior, however, remains transparent to the developer, as it does not matter where the space is located physically. The data-driven approach allows the design of P2P architectures without the need to manage communication between peers directly. Therefore, a shared blackboard combines the flexibility of P2P systems with the simplicity of a single interaction partner for each peer.

The space-based computing paradigm (SBC) was introduced by David Gelernter in the form of the Linda tuple space [13]. In this model, information is shared via the space in the form of tuples, which are sequences of typed fields. Tuples can be written to the space as well as read or consumingly read from it by providing a template that is compared to existing tuples in the space. The read operations may optionally block while no appropriate tuple can be found. All fields defined in the template must match exactly with the corresponding fields in the tuple, but some fields may be undefined, which are interpreted as wildcards that allow any value. In case that more than one tuple is found, one is selected indeterministically. This mechanism named template matching allows for a simple coordination among peers by reading a particular tuple with a template that leaves the needed data fields undefined. If the tuple is provided by a peer, the read operation succeeds and the first peer gets its requested data. A well-known representative of the Linda model is JavaSpaces [12], which is defined as a Java language binding. Instead of tuples, a JavaSpace stores entry objects, which correspond to Linda tuples that are automatically derived from the public member variables of the respective entry class. The basic operations are `read`, its consuming variant `take`, `write` and `notify`, which can be seen as an asynchronous query on future write operations, where a callback method is called as soon as a matching entry is written to the space. The read and take methods

contain a timeout parameter that specifies how long the call may block until it returns with an error. JavaSpaces also allows transactions for its operations, which enables the programmer to encapsulate several space operations into a single atomic action. With this coordination model, an effective collaboration between several autonomous peers can be established in a simple way, but nevertheless JavaSpaces and its Linda foundation have various drawbacks:

- If several entries match a template, subsequent reads cannot guarantee that all matching entries are retrieved eventually. Due to the indeterministic matching mechanism, the read operation could also return the same entry every time, although the application might be interested in all relevant entries. To solve this problem, the programmer has to issue several take operations until no matching entries are found, thus preventing other processes from accessing these data.

- The selection of entries is based on the exact matching of the fields' values. A query on a certain range of allowed values for a field is therefore not possible.

- No information about the chronology of write operations is stored, so a simple first-in-first-out ordering of tuples can only be generated by adding and managing own sequence numbers.

- Complex coordination laws that cannot easily be mapped to template matching must be enforced by the application programmer, so that fields with user and coordination data are mixed together in entries, accompanied by several extra entries that are just used for coordination purposes. E.g., the implementation of a simple producer/consumer pattern requires sequence numbers for all written entries and separate counter tuples for the highest sequence number and the number of the entry that should be consumed next.

- Entries may only match a template of the same type (or of a super type), so they must have the same count and order of fields. It is not possible to search for entries with a particular value that can occur in several different entry types. Instead the read operation must be issued for every entry type with separate templates.

- JavaSpaces can only be used for Java applications. GigaSpaces [9], which is a wide-spread JavaSpaces implementation, offers also language bindings for C++ and .NET, but no open protocol has been published that would allow interoperability between arbitrary language adapters.

Over the past years, several space-based computing technologies based on the original Linda model have evolved, which try to overcome the mentioned disadvantages. A

different space-based middleware approach has been established with the virtual shared memory model of Corso [23], developed at the Institute of Computer Languages of the Vienna University of Technology. In contrast to Linda, shared objects are identified via unique identifiers called OIDs that can be found via well-known names. These OIDs can be linked together, which enables complex and highly concurrent coordination patterns. Furthermore, Corso supports distributed transactions, replication, persistency and notifications.

Based on the experiences with Linda-like tuple spaces and Corso, the concept of XVSM[1] [27] (eXtensible Virtual Shared Memory) has been developed at the Institute of Computer Languages of the Vienna University of Technology. This thesis provides a formal definition of this system as a foundation for further research and development. The specification is also implemented in a functional language as an executable prototype, which serves both as a reference for developers of further XVSM implementations and as starting point for formal verification of the model.

The rest of this introduction covers the basic concepts of XVSM and the current status of its implementation as well as the reasons for the development of a formal model and related work. Section 2 explains the basic algebraic data types and operations that are used in the description of the XVSM model. Section 3 introduces the basic query language to select elements within the space. The actual formal specification of XVSM is shown in Sect. 4, where the space operations are bootstrapped in a layered architecture consisting of several core APIs. Section 5 presents an example for the efficient utilization of XVSM in complex scenarios. In Sect. 6, the prototype implementation in the functional programming language Haskell is presented. Section 7 describes the influence of the formal model on future implementations of the XVSM middleware and outlines open issues and topics for further research. Finally, Sect. 8 gives a conclusion of this thesis.

1.1 XVSM Basics

The main objective of the XVSM middleware is to provide a flexible, extensible and easy-to-use platform for the collaboration of processes, which are dynamically joining and leaving, on heterogeneous platforms. The space holds shared data of these processes which can be used to exchange information, to issue a request for a certain task and provide the corresponding reply, or for synchronization purposes. An XVSM space may run in embedded mode within a peer application or as a stand-alone server. For the application programmer, however, there should be no difference between access to a local or a remote space, as the XVSM core hides the network communication.

Data is encapsulated in entries that are grouped in containers, which can be seen

[1] `http://www.xvsm.org`

as sub spaces with own coordination mechanisms. Similar to JavaSpaces, it is possible within a container to write, read and take entries and set notifications on future events. The coordination law for extracting entries from a container is however flexible and not limited to simple template matching like in the Linda model. Each container can be associated with one or more coordinators that are responsible for the bookkeeping of the entries. When a read or take request is issued, the involved coordinators select the resulting entries. Depending on the coordinator, the selection could be based on a FIFO queue, a certain key value given by the user, the order of entries regarding a given data field, or template matching. Also, more complex coordination mechanisms can be defined by the user and even a combination of several coordinators is possible for a single query. As in JavaSpaces, any read or take operation may block until the respective query is fulfilled or the specified timeout is expired. In XVSM, however, also write can block, e.g. when a bounded container is full.

Transactions enable the programmer to combine several space operations to a single atomic action, which is helpful for complex concurrent applications. It is also possible to alter the semantics of any operation by dynamically adding aspects that are executed before or after each invocation, respectively. These aspects may change the operation's parameters, manipulate its return values, execute accompanying tasks like logging the operation to a special container or to an external file, or even skip the original operation. With these mechanisms, additional profiles can be added to the space implementation, which fulfill tasks like lookup, security, monitoring, persistency and replication.

In contrast to JavaSpaces, the XVSM middleware is not defined as a language binding, but instead as a language independent XML protocol called XVSMP. The protocol supports synchronous operations as well as asynchronous variants where the result is given to a callback method or written to an answer container that the caller can access at a later time. Entities in a space like containers, transactions and aspects are referenced via a unique URI consisting of the space URI and a local identifier. On top of this protocol, language bindings for arbitrary platforms can be developed quickly, so that applications written in different programming languages may access the same space. Due to the standardized protocol, different XVSM core implementations can also interoperate with each other.

The original design of XVSM was created in 2005 by Kühn et al. [27]. Since then, a lot of effort has been made to enhance this middleware architecture and to develop stable and efficient implementations of the XVSM core and of additional profiles. Several use cases for the application of the XVSM middleware have been presented since its introduction, like in the production automation domain [24, 25] or in rescue scenarios [28]. Currently, there exist two implementations of XVSM: The open-source Java imple-

mentation MozartSpaces[2] [44, 42], and XcoSpaces [43, 22], which is written for the .NET platform in C#. These versions are used in several educational and commercial projects and are also able to interoperate with each other with some restrictions. XVSM versions for other platforms are also in development, like for the .NET Micro Framework[3], which is used in small embedded devices [30]. With the development of the XVSM formal model presented in this thesis, it is necessary to adapt current implementations to the modifications in the design of the middleware while simultaneously improving performance, stability and usability. Therefore, a new version of MozartSpaces is currently in development, which should allow the testing of the improved architecture in real scenarios. Subsequently, implementations on other platforms will be updated to reflect the new semantics.

1.2 Motivation for a Formal Model

The goal of this work is to develop a formal model of XVSM that specifies its behavior in a detailed way. Otherwise, different XVSM implementations would behave differently under some circumstances, especially in concurrent situations and in the case of errors. This would complicate the development of portable and interoperable applications with the space middleware. Therefore, the formal model should serve as an unambiguous reference for implementations of XVSM on various platforms and in different programming languages. The formalization process is also helpful in the design phase of the middleware architecture because conceptual errors can be recognized earlier if the middleware's behavior can be analyzed exactly before the actual implementation. The model should provide clear semantics of all of the middleware's operations and also describe their limitations. However, this foundation does not cover implementation issues like performance optimizations or how data is stored internally. These problems are left to the developers of XVSM implementations that may exploit special techniques available on their platforms. The formal model gives a definition of how the middleware reacts to a certain input in a given scenario, which must be followed by all XVSM implementations. Although the model also offers a complete specification of the data structure of the space and its internal operations (see Sect. 4), the middleware developers are not forced to obey this reference as long as their implementation has the same semantics as the formal model.

The reference model can not only be used as a basis for developers of XVSM implementations, but also as a foundation for theoretical research on the middleware's traits. It should be an origin for verification of certain properties of the space, which is needed for the standardization and certification of XVSM. Only if it is possible to prove the

[2] `http://www.mozartspaces.org`
[3] `http://www.microsoft.com/netmf/`

correctness of the middleware's key concepts, it will be accepted by industrial partners in safety-critical scenarios.

The focus of this thesis is to specify the semantics of the XVSM core, which executes user operations on a single space and communicates with other cores for remote interactions. The exact definition of aspects, the XML protocol and profiles with additional features is out of scope of this work, thus only an outline of these components is given. The formal model is specified as a hierarchy of core API (= CAPI) layers (see Sect. 4) based on a simple algebra described in Sect. 2. The CAPI operations are specified as a combination of textual definitions and pseudo-code. Operational semantics that use mathematical formalism, however, might be used in future work to enable easier verification of XVSM. As a proof of concept of the XVSM definition, a prototype of the XVSM core has been written in the functional programming language Haskell (see Sect. 6), which complies with the formal model as much as possible. This enables the immediate testing of changes to the model in a practical environment.

The basic outline of the XVSM formal model has been published in [26] and [10]. This thesis continues the research on this topic and elaborates the concepts given in these papers, as well as introducing the Haskell reference implementation of the XVSM core.

1.3 Related Work

Formalization is widely used to model software architectures for complex distributed applications. Especially for middleware systems, it must be ensured that developers can rely on well-defined semantics. In [6] and [5], which specify core concepts of JavaSpaces, the authors use formal methods to remove ambiguity from the informal semantics definition, to analyze the expressiveness of the middleware compared to similar systems and to enable verification of programs using JavaSpaces. The analysis of a formal model may also lead to improvements and extensions for already-in-use systems, as described in [31] for the implementation of a Web Coordination Service based on JavaSpaces. According to [41], the static parts of a system can be expressed with algebraic data types, while for the dynamic behavior, several different formal approaches are feasible, including process algebras and temporal logic.

With process algebras, also known as process calculi, concurrent systems can be specified in a formal way. They describe the interaction of independent processes using a limited amount of primitives. Algebraic laws are used to transform these specifications into different expressions, thus allowing for formal reasoning. One widely used process calculus is CSP (Communicating Sequential Processes) [17] by C. A. R. Hoare. The abstract interaction between systems and their environments is described with a math-

ematical formalism. Processes, which may run sequentially or in parallel, communicate by exchanging messages. By using recursion, the calculus is able to describe processes that run infinitely. A similar approach is used by Robin Milner's π-calculus [32], which relies on names and a very small set of operators to model concurrent computations. Another possibility to define the semantics of programs is with the Structural Operational Semantics (SOS) [40] invented by Gordon Plotkin. Based on the abstract syntax of a program, this method specifies its behavior with inference rules that define the possible transitions for a program from the valid transitions of its syntactical components. Thus, the rules specify how a complex program must react when some part of the program fulfills a certain condition for a specific program state. A different approach to express the semantics of concurrent systems is via temporal logic, as shown with TLA (Temporal Logic of Actions) [29], which was developed by Leslie Lamport. Temporal logic combines ordinary logical operators like conjunction, negation and implication with modal operators that specify the temporal relations between the conditions. In TLA, both the algorithm and the program properties that need to be verified are specified in a single logic formalism, which is then used for reasoning.

Also some work has been published on the formalization of space-based computing middleware. The operational semantics of the classical Linda model can be expressed with a process algebra using traces [14], which are sequences of valid Linda actions. The formal model enables the specification of the communication mechanisms and the blocking behavior of Linda in an unambiguous way. In [4] and [3], a simple process calculus termed LinCa is used to show that the expressiveness of the Linda model can be extended if tuples include quantitative labels for probabilities and priorities. The authors of [6] and [5] develop a process algebra for Linda and then extend it to depict the semantics of notifications, timeouts and leasing, thus incrementally creating a formal model for an abstract language similar to JavaSpaces.

Another space-based coordination model defined via a formal model is PoliS [7]. In PoliS, a space can contain sub spaces that include ordinary data as well as program tuples. The programs within the space can modify this tree data structure when their corresponding precondition is fulfilled, thus the structure of the space is constantly changing. A program includes a rule that specifies how the tuples of the space shall be rewritten. The formalism used to define the operational semantics is SOS, whereas TLA is applied for the reasoning on the coordination architectures that are modeled with PoliS.

The KLAIM system [35] provides a programming language for mobile processes, realized via multiple Linda tuple spaces and an additional operation set that supports the management of processes. Mobile agents can be moved within the KLAIM net from one system node to another, just like ordinary data. The integrated type system enforces security properties by checking that processes do not violate any access rights. The pro-

gramming language syntax itself is based on process algebras, thus KLAIM presents an asynchronous higher-order process calculus. The operational semantics of KLAIM are defined using Plotkin's SOS approach.

In [34], a knowledge-based coordination model for subscriptions on a semantic repository is presented. Using temporal logic, the semantics of the notification mechanism is defined with the help of safety and liveness conditions. The safety condition specifies when a notification may be fired, while the liveness condition guarantees that a subscriber must eventually receive a notification if a matching event occurs.

The XVSM middleware itself remains without a formal foundation until now. A comparison to JavaSpaces, however, shows the necessity of a formal model. The concise JavaSpaces specification [47] is not able to cover all semantic details. Such informal definitions are usually too ambiguous and leave much freedom in implementation choices, which leads to different implementations with varying semantics [6, 5]. Therefore, this thesis presents a detailed specification for XVSM, which can later be transformed into a strictly formal model using process calculi, temporal logic or similar approaches.

2 XVSM Algebra

The formal description of XVSM is based on a simple algebraic foundation [10] that defines the middleware's basic data structures and how they can be accessed and manipulated. In general, the space consists of a nested hierarchy of data collections that contain objects like strings or integers, as well as other collections. These data items, which can be arranged in a sequential list or in an unordered bag, have a label to identify them. Sets, which require all members to be distinct, are not used in the formal model because they would need complex equality checks if items are added to them. Instead, sequences (= lists) are used if the ordering of the labeled values is important, and multisets (= bags), which are basically sets that allow duplicates, otherwise. Thus, the basic data structure is a tree of labeled multisets and sequences with labeled values as leaves. This structure is called an xtree, which can be defined recursively as follows:

Definition: An xtree is either a sequence or a multiset of labeled xtrees, or an unstructured value like a string or an integer.

A multiset xtree can be written as $[l_1{:}x_1, l_2{:}x_2, \ldots]$, where the ls are labels and the xs are xtrees. Similarly, a sequence xtree is represented as $\langle l_1{:}x_1, l_2{:}x_2, \ldots \rangle$. If an xtree does not have to be accessed directly, it does not need an explicit label, thus the empty or anonymous label can be used. As a matter of convenience, $:x$ may be abbreviated as x if the xtree cannot be mistaken as a label. Another possible shortcut is to write only the label l instead of l:true. As an example, consider the following sequence:

$$X = \langle \texttt{abc}, \texttt{a}{:}24, \texttt{a}{:}42, \texttt{b}{:}[\texttt{pi}{:}3.14, \texttt{e}{:}\text{“text”}], [\,] \rangle$$

This xtree consists of an implicit Boolean value `true` labeled with `abc`, two integer values with the same label `a`, a multiset `b` containing a float (`pi`) and a string (`e`), and an empty multiset implicitly tagged with the anonymous label. To navigate within an xtree, the labels of the xtrees are used. The path identifying a sub xtree consists of all xtree labels from the root to the searched xtree. The labels within an xtree do not necessarily have to be distinct, but the navigation becomes indeterministic if they are not. For sequence xtrees, the index number can also be used for accessing a child xtree instead of using a label. Formally, a path is a sequence of labels or indices separated by slashes. To access a substructure of an xtree, a period and the path to the searched xtree are appended to it. The xtree selected by a path is recursively defined as follows, where

notation	*meaning*	*notation*	*meaning*
$[\,]$	empty multiset	$\langle\,\rangle$	empty sequence
$B \sqcup B'$	union	$C \cdot C'$	concatenation
$B = B'$	multiset equality	$C = C'$	sequence equality
$B \sqsubseteq B'$	submultiset relation	$a \in C$	sequence membership
$a \in B$	multiset membership	$\lvert C \rvert$	length
$\lvert B \rvert$	cardinality	$\text{first}(C)$	first element
$\text{any}(B)$	element selection	$\text{rest}(C)$	all except first
$B - B'$	multiset difference	$\text{nth}(n, C)$	n-th element of sequence

Table 1: Operations on multiset and sequences

ϵ denotes the empty element that implicitly terminates each path:

$$
\begin{aligned}
x.\epsilon &= x \\
[\ldots, l{:}x, \ldots].(l/p) &= x.p \quad \text{for any label } l \\
\langle \ldots, l{:}x, \ldots \rangle.(l/p) &= x.p \quad \text{for any label } l \\
\langle l_1{:}x_1 \ldots, l_i{:}x_i, \ldots \rangle.(i/p) &= x_i.p \quad \text{for any valid index number } i
\end{aligned}
$$

For example, the following expressions are valid for the xtree defined above: $X.\texttt{b} = X.4 = [\texttt{pi}{:}3.14, \texttt{e}{:}\text{“text”}]$, $X.(\texttt{b}/\texttt{pi}) = 3.14$ and $X.5 = [\,]$. The expression $X.\texttt{a}$ may either be 24 or 42, of which one is chosen indeterministically. The evaluation of the access based on the previous definition is shown in the following:

$$X.(\texttt{b}/\texttt{e}) = X.(\texttt{b}/\texttt{e}/\epsilon) = [\texttt{pi}{:}3.14, \texttt{e}{:}\text{“text”}].(\texttt{e}/\epsilon) = \text{“text”}.\epsilon = \text{“text”}$$

For a meaningful usage of this data structure for the definition of the XVSM formal model, it is necessary to also specify operations on xtrees. Basically, all functions on multisets and sequences, like union, difference or cardinality, can be used. Table 1 shows examples for possible operations.

This basic data structure is used to define the XVSM formal model. Basically, a *space* is a multiset xtree that comprises all *containers* located at a single site, together with their unique labels that serve as their references. A container is itself a multiset xtree containing user-generated *entries* with unique labels. Entries consist of labeled values called *properties*, which have a well-defined label and a value that can either be an unstructured value like an integer or string, or a more complex xtree. Properties correspond to the attributes of the object represented by the entry. An entry may represent a real-life object, a message with header and payload attributes, or a command with the arguments and quality parameters modeled as properties. An entry depicting a book might look as

follows:

$$[\texttt{title}:\text{“JavaSpaces”}, \texttt{author}:\text{“Freeman”}, \texttt{author}:\text{“Hupfer”}, \texttt{author}:\text{“Arnold”},$$
$$\texttt{date}:1999, \texttt{publisher}:\text{“Addison-Wesley”}, \texttt{nPages}:368]$$

As seen in the example, properties with the same label are allowed within entries. However, the book entry could also be modeled with unique labels if this is more appropriate:

$$[\texttt{title}:\text{“JavaSpaces”}, \texttt{author}:\langle \texttt{a1}:\text{“Freeman”}, \texttt{a2}:\text{“Hupfer”}, \texttt{a3}:\text{“Arnold”}\rangle,$$
$$\texttt{date}:1999, \texttt{publisher}:\text{“Addison-Wesley”}, \texttt{nPages}:368]$$

As explained above, the XVSM space and all of its components can be seen as xtrees. When formalizing the interaction between several spaces, the set of all existing spaces can be modeled in one single xtree called the *universe*. Therein the space xtrees are labeled with their unique URI.

For the rest of this thesis, labeled values within xtrees will be called properties, even outside of user entries. Beside the actual user data stored in the entries of a container, there are also meta properties in the XVSM data structure that are accessed by different CAPI layers as an internal storage to bootstrap the behavior of the space. Meta data can be stored in simple meta properties at space, container and entry level, or in dedicated meta containers. Examples for meta properties are transaction locks, coordinator information like a key or a vector position of an entry, or the sequence of scheduled requests within the runtime machine. Meta data are identified by labels starting with an underscore (_). Together, meta and user data form an xtree that depicts the complete state of an XVSM space during runtime. The data structure of this xtree is defined in the so-called *XVSM meta model*. Each CAPI layer accesses different meta properties via their well-known path. Meta data can be manipulated by the same mechanisms as regular user data, namely with read, take and write operations. The exact structure of the meta model is defined in Appendix A, while the usage of the meta properties is explained in Sect. 4.

3 XVSM Query Language

3.1 Query Structure

To utilize the algebraic data structure for modeling the whole space functionality in an efficient way, there must be more advanced access mechanisms than selection by name or by position. Therefore, a basic query language is used that enables the selection and reordering of specific xtrees. This *XVSM Query Language* (XVSMQL) can be applied by the user of the space to select entries from a container, but it is also used on arbitrary xtrees of the XVSM meta model for the bootstrapping of the core API specification. The following paragraphs describe the general semantics of the query language when applied on a given sequence or multiset xtree, while Sect. 4.1 describes how a query is evaluated when used as an argument of a read or take operation.

An *XVSM Query* (XQ) consists of one or more predicates called *Simple XVSM Queries* (SXQ) that are chained via the pipe operator. Therefore, the query has the following general form:

$$SXQ_1|SXQ_2|\ldots|SXQ_n$$

A query evaluation on a given xtree is started by applying the first SXQ on the whole xtree, which yields a multiset or sequence xtree of properties fulfilling the predicate. This resulting xtree is then used as an input for the next SXQ and so forth, until the last SXQ returns the result xtree of the entire query. From stage to stage, the cardinality of the result can only decrease or stay the same, as the predicates may only select or reorder the properties but may not add further data. An XVSM query can thus be seen as a filter mechanism that only allows those properties to pass through that fulfill all predicates specified by the SXQs.

An SXQ can be classified into two basic categories, which are *matchmakers* and *selectors*. A matchmaker is a predicate that can be tested for each single element of the given xtree and evaluates unambiguously to either true or false. The properties that fulfill the predicate remain in the result xtree whereas the others are cut out, while the relative order of the remaining xtrees stays the same in case of a sequence. A selector, on the other side, is always evaluated on the whole input xtree because the function may not only filter certain elements but also change their order, thus it must be able to compare all existing elements. Selectors can also change an unordered multiset input xtree into a sequence result xtree and vice versa. To start evaluation of a selector SXQ, the entire input xtree must be available while matchmakers could be applied on partly existent inputs because they are evaluated on the properties one by one. Consecutive matchmaker SXQs are therefore commutative and can be evaluated in any order and even in paral-

lel, which offers possibilities for optimizations. However, when a selector SXQ occurs in a filter chain, the computation of all previous SXQs must be finished as the selector's evaluation depends on the entire xtree and cannot be started with incomplete input. Formally, a matchmaker function is a mapping from the set of all valid properties to a Boolean value. A selector function transforms a sequence or multiset xtree into another sequence or multiset xtree, whereas the set of properties within the result xtree must be a subset of the properties occurring in the input. Both function types may optionally also have arguments of arbitrary types which specify how the concrete SXQ should be applied to the input. In contrast to a matchmaker, a selector function may also fail with a well-defined error if its selection criteria or order preconditions are not fulfilled for the given input. As long as a function complies with this specification, it can be used as an SXQ, making this query language extensible. In general, however, most queries consist only of a limited amount of predefined query predicates, which are introduced in Sect. 3.2. Additionally, more specialized matchmaker and selector functions will be specified in this thesis when needed.

3.2 Predefined Queries

Using only the predefined query elements presented in this section, it is already possible to express many different selection conditions. In the following, the selector and matchmaker functions of XVSMQL will be described by specifying their syntax and semantics. To illustrate the behavior of these query predicates, suitable examples are given using the following example entries representing books:

$E_1 =$ [`title`:"JavaSpaces", `author`:"Freeman", `author`:"Hupfer", `author`:"Arnold", `publication`:[`date`:1999, `by`:"Addison-Wesley"], `nPages`:368]

$E_2 =$ [`title`:"Virtual Shared Memory for Distributed Architectures", `author`:"Kühn", `publication`:[`date`:2001, `by`:"Nova Science Pub Inc"], `nPages`:112]

$E_3 =$ [`title`:"JavaSpaces in Practice", `author`:"Bishop", `author`:"Warren", `publication`:[`date`:2002, `by`:"Addison-Wesley"]]

$E_4 =$ [`title`:"JavaSpaces Example by Example", `author`:"Halter", `publication`:[`date`:2002, `by`:"Prentice Hall PTR"], `nPages`:400]

$E_5 =$ [`title`:"How to write parallel programs: a first course", `author`:"Carriero", `author`:"Gelernter", `publication`:[`date`:1990, `by`:"MIT Press"]]

These entries contain a title, a variable amount of author properties, a publication property containing a nested multiset with date and publisher information and optionally also a property with the number of pages. The variables E_1 to E_5 are used to abbreviate the representation of the container. There are two different versions of the input xtree that contains these five entries, namely one multiset X_{Ms} and one sequence X_{Seq}:

$$X_{\text{Ms}} = [E_1, E_2, E_3, E_4, E_5]$$
$$X_{\text{Seq}} = \langle E_1, E_2, E_3, E_4, E_5 \rangle$$

If both xtrees return the same result for a query, they are simply abbreviated with X.

3.2.1 Predefined Selectors

Count operator

- $cnt(n)$

The `cnt` predicate takes the first `n` entries of the input, if it is a sequence, else `n` entries are selected from the multiset arbitrarily. The given argument must be an integer value greater than 0. If there are not enough entries available, the query fails.

Examples:

$$X_{\text{Seq}} \mid cnt(3) = \langle E_1, E_2, E_3 \rangle$$
$$X_{\text{Ms}} \mid cnt(1) = [E_3]$$
$$X \mid cnt(9) = \text{ERROR!}$$

Note: For multisets, the selection of entries is indeterministic, so instead of E_3, any other entry could be selected.

Sorting

- $sortup(p)$
- $sortdown(p)$
- $reverse()$
- $id()$

The `sortup` and `sortdown` filters sort the entries in ascending or descending order of the property values at the specified relative path `p` within the entry as defined in Sect. 2.

The path may contain wildcards (*) matching all labels, so a value of `*/date` would match the relative path `publication/date` as well as any other nested `date` values in one of the entry's properties. The sorting is stable and entries without the property are appended at the end. If an entry has several properties at the given path, one is chosen indeterministically for sorting. It is assumed that the compared values have a natural order, otherwise the function returns an error. The sorting functions always return a sequence xtree as result, even if the input is a multiset. The `reverse` function inverts the internal order of the input sequence and returns an error if used on a multiset xtree, while the `id` selector simply returns the input xtree without any changes.

Examples:

$$
\begin{aligned}
X \mid \mathit{sortup}(\text{*/date}) &= \langle E_5, E_1, E_2, E_3, E_4 \rangle \\
X \mid \mathit{sortup}(\text{author}) &= \langle E_3, E_5, E_1, E_4, E_2 \rangle \\
X \mid \mathit{sortdown}(\text{nPages}) &= \langle E_4, E_1, E_2, E_3, E_5 \rangle \\
X \mid \mathit{sortdown}(\text{publication/by}) &= \langle E_4, E_2, E_5, E_1, E_3 \rangle \\
X_{\text{Seq}} \mid \mathit{reverse}() &= \langle E_5, E_4, E_3, E_2, E_1 \rangle \\
X_{\text{Ms}} \mid \mathit{reverse}() &= \text{ERROR!} \\
X_{\text{Ms}} \mid \mathit{id}() &= [E_1, E_2, E_3, E_4, E_5]
\end{aligned}
$$

Note: As the author property is ambiguous for some entries, the corresponding sorting query may also have different solutions, dependent on the indeterministically chosen property used for comparison.

Uniqueness

- $\mathit{distinct}(p)$

This filter creates an output xtree that has unique values for properties at the specified relative path `p` within the entry, which may also contain wildcards. In case that more than one entry has the same property value, the first one is taken for sequences, or any one in case of a multiset input xtree. If an entry has several properties at the given path, one is chosen indeterministically for comparison. Entries that do not have a property at the given path are not included in the result.

Examples:

$$X_{\text{Seq}} \mid \mathit{distinct}(\text{publication/date}) = \langle E_1, E_2, E_3, E_5 \rangle$$
$$X_{\text{Ms}} \mid \mathit{distinct}(\text{publication/by}) = [E_1, E_2, E_4, E_5]$$
$$X_{\text{Seq}} \mid \mathit{distinct}(\text{nPages}) = \langle E_1, E_2, E_4 \rangle$$

Note: For multisets, the selection of entries is indeterministic, so E_3 could also be included in the result instead of E_1 when selecting unique publishers.

3.2.2 Predefined Matchmakers

Property predicates

- $p \in \{rangeExpr\}$
- $p \notin \{rangeExpr\}$
- $\forall p \in \{rangeExpr\}$
- $\forall p \notin \{rangeExpr\}$

Every property can be checked against a set of allowed or forbidden values by using a filter of the form $p \in rangeExpr$ and $p \notin rangeExpr$, respectively. It is assumed that all values have a predefined ordering, so that `rangeExpr` can be any set of values or intervals. The specified path `p` denotes which (sub) xtree of the given entry is actually compared to the range. In contrast to the relative entry paths used for the previously defined selectors, the path is relative to the whole input xtree and not the child xtrees, so that the labels of the entries can also be involved in the query. This allows selecting meta properties with a certain name from an xtree. E.g., a path of `l1` denotes the entry with the corresponding label itself. Usually, queries are issued on properties of an entry, which is written as `*/propLbl1`, meaning that for any entry (with arbitrary label) the query must look at the property labeled with `propLbl1`. It is also possible to query sub properties, so `*/*/subLbl1` or `*/propLbl2/subLbl2` are valid paths. The expression `rangeExpr` has the following syntax, specified in EBNF notation [21]:

$$\begin{aligned} rangeExpr &= \text{“..”} \mid expr\ \{\text{“,”}\ expr\} \\ expr &= val \mid lower\ \text{“..”}\ upper \mid lower\ \text{“..”} \mid \text{“..”}\ upper \\ lower &= val \mid val\ \text{“>”} \\ upper &= val \mid \text{“<”}\ val \end{aligned}$$

In this description, `val` stands for any valid value of the property. It is possible to select any value ($\{..\}$), a set of values (e.g. $\{v_1, v_2, v_3\}$) or a set of intervals like $\{..4, 7..11, 13..\}$,

which denotes any values less than or equal 4, greater than or equal 13 or between 7 and 11 whereas both boundaries are included. Also, a combination of single values and intervals is allowed. For open intervals, the predecessor ($< val$) and successor ($val >$) are used. A range expression like $\{12 > .. < 24\}$ includes any values greater than 12 and less than 24, so the boundaries are not included here.

The matchmaker selects all entries with at least one property at path `p` with a value that is (not) contained in the given range. Due to duplicate labels and the use of wildcards, a path can determine several different properties. For the evaluation of the predicate, it is only necessary to find one property value that fulfills the expression. So, to determine that a property does not fulfill the matchmaker, all fitting properties must be examined. If, however, a universal quantifier ($\forall$) is used with the path, all properties at the given path must comply with the expression, otherwise the entry is not included in the result xtree. If no property at path `p` exists, the entry is also not included.
There are also some convenience operators defined which are based on the preceding predicates:

- $p = val$, $p \neq val$
- $p \leq val$, $p < val$, $p \geq val$, $p > val$
- $\forall p = val$, $\forall p \neq val$
- $\forall p \leq val$, $\forall p < val$, $\forall p \geq val$, $\forall p > val$
- p

All of these predicates can be easily transformed into the previously defined extensive syntax. E.g., $p = val$ corresponds to $p \in \{val\}$, whereas $p < val$ is equivalent to $p \in \{.. < val\}$. If a predicate only consists of the path p, all entries containing a property with arbitrary value at the given path are selected. In the extensive syntax, this corresponds to $p \in \{..\}$.

It must be noted that with these semantics, certain conflicting operators are not mutually exclusive when used with the same arguments. E.g., $p = val$ and $p \neq val$, or $p < val$ and $p > val$ can be true for an entry at the same time if the path denotes several different properties. To guarantee that at most one of these predicates is true at any time, the universally quantified versions must be used.

Examples:

$$(X_{\text{Seq}} \mid \text{*/author} = \text{“Gelernter”}) = \langle E_5 \rangle$$
$$(X_{\text{Seq}} \mid \text{*/author} \neq \text{“Gelernter”}) = \langle E_1, E_2, E_3, E_4, E_5 \rangle$$
$$(X_{\text{Seq}} \mid \forall\ \text{*/author} \neq \text{“Gelernter”}) = \langle E_1, E_2, E_3, E_4 \rangle$$
$$(X_{\text{Ms}} \mid \forall\ \text{*/author} \geq \text{“C”}) = [E_2, E_4, E_5]$$
$$(X_{\text{Ms}} \mid * = E_1) = [E_1]$$
$$(X_{\text{Seq}} \mid \text{*/publication/date} \in \{1990 > ..1995\}) = \langle\,\rangle$$
$$(X_{\text{Seq}} \mid \text{*/publication/by} \notin \{\text{“Addison-Wesley”}, \text{“MIT Press”}\}) = \langle E_2, E_4 \rangle$$
$$(X_{\text{Ms}} \mid \text{*/nPages}) = [E_1, E_2, E_4]$$

Logical operations

- $A \wedge B$
- $A \vee B$
- $\neg A$

All matchmakers can be combined with logical conjunction, disjunction and negation. When using more than one logical operator in an expression, the following precedence rules hold: Negation has higher precedence than conjunction, which has higher precedence than disjunction. To change this default behavior, parentheses must be used. A property predicate using a range expression can also be written as a disjunction. E.g., */author $\in$ {“Gelernter”, “Kühn”} is equivalent to */author = “Gelernter” $\vee$ */author = “Kühn”. Two consecutive matchmaker functions in the query filter chain could always be replaced by a single matchmaker formed by the conjunction of both SXQs. Formally, the logical operations are higher-order functions that take one or two other matchmakers as arguments to form a new matchmaker function that can be evaluated on the entries.

Examples:

$$(X_{\text{Seq}} \mid \text{*/author} = \text{“Gelernter”} \ \wedge\ \text{*/author} = \text{“Carriero”}) = \langle E_5 \rangle$$
$$(X_{\text{Seq}} \mid \text{*/nPages} \ \vee\ \text{*/publication/date} < 1999) = \langle E_1, E_2, E_4, E_5 \rangle$$
$$(X_{\text{Seq}} \mid \neg\ \text{*/author} = \text{“Gelernter”}) = \langle E_1, E_2, E_3, E_4 \rangle$$
$$(X_{\text{Ms}} \mid (\text{*/nPages} < 400 \ \vee\ \text{*/author} = \text{“Bishop”}) \wedge \neg\ (\text{*/publication/by} = \text{“Addison-Wesley”})) = [E_2]$$

3.2.3 Combined Queries

The previously defined SXQs can be combined in an arbitrary way to form a complex query, as shown in the following examples:

Get all books from publisher "Addison-Wesley" sorted by publication date:
$(X \mid */\text{publication/by} = \text{“Addison-Wesley”} \mid \mathit{sortup}(\text{publication/date})) = \langle E_1, E_3 \rangle$

Get the newest book that has over 300 pages:
$(X \mid */\text{nPages} > 300 \mid \mathit{sortdown}(\text{publication/date}) \mid \mathit{cnt}(1)) = \langle E_4 \rangle$

Of all the books published in 2002, get the one with the second title in lexicographical order:
$(X \mid */\text{publication/date} = 2002 \mid \mathit{sortup}(\text{title}) \mid \mathit{cnt}(2) \mid \mathit{reverse}() \mid \mathit{cnt}(1) = \langle E_3 \rangle$

3.3 Analysis and Comparison

Initially, the query language was planned to be used only on entries within user containers. This implies that for access and manipulation of the meta model, separate methods would have to be introduced to read, set and delete meta properties by specifying their unique path. To simplify the formal model, XVSMQL has been defined for all xtrees instead of only for user containers. Thus, every operation on the space can be mapped internally to a series of simple read, take and write operations on the meta model, as described in Sect. 4. This decision to use the query language also for the bootstrapping of the XVSM runtime has influenced the specifications of the predefined SXQs. The use of wildcards in the path is usually not necessary for user entries as these have a well-defined structure and the leading wildcard indicating an arbitrary label can be implicitly assumed. In this case, the matchmaker */nPages > 300 would simply be written as nPages > 300. To retrieve arbitrary data within the meta model hierarchy, however, it is feasible to allow wildcards because many labels in this model are auto-generated and it would be tedious to issue a query to determine the property labels before the actual query can be executed. Therefore, the introduction of wildcards allows for more flexibility and easier queries on the meta model.

The XVSMQL is loosely based on a subset of SQL SELECT statements, which are used for queries in relational databases. An input xtree in XVSM can be compared to a single table in a database, as both contain structured data. The main differences are that database tables have records with a flat structure and fixed fields, while xtrees may inhibit inhomogeneous and possibly nested data. For the definition of the query language, however, this is not a big issue because the nested structure can simply be

expressed by paths instead of names, and entries that do not contain a certain property are treated as if the property implicitly has the value `null`. In contrast to SQL, where joins between different tables are possible, XVSMQL only works on a single input data structure. As XQs only filter entries without changing their appearance, projections (i.e. only returning certain properties of matching entries) and aggregate functions that return values computed from the result (like sum, average or count) are not possible. Basically, an XQ can be compared to an SQL query of the form `"SELECT * FROM xt WHERE ... ORDER BY ..."`, where `xt` is either the identifier of the original input data or a nested subselect statement of the same form. The operators within a `WHERE` clause are very similar to the predefined matchmakers in XVSMQL: If compared with literal values (i.e. no comparison with other data fields or subselect results), the SQL predicates `IN`, `BETWEEN`, `IS NULL` as well as the common comparison operators `=`, `<>`, `>`, `>=`, `<` and `<=` can be directly mapped to XVSMQL property predicates. Also, the logical predicates are semantically equivalent. Only the `LIKE` predicate, which determines if a string fits a certain pattern, is not supported in XVSMQL. The `ORDER BY` clause, which sorts the result set by one or more fields, corresponds to a combination of one or more XVSMQL sorting selectors. The `distinct` selector, however, has a slightly different meaning as the corresponding keyword in an SQL `SELECT` clause. In SQL, only the column marked as distinct can be displayed in the result set, whereas XVSMQL returns the whole entries with all properties. A possible counterpart of the `cnt` selector in SQL is the non-standard `LIMIT` clause, which allows to specify a maximum amount of returned rows. There is, however, no error message if not enough elements are found.

Another query language that has comparable expressiveness is the one used by JMS *message selectors* [48]. With JMS, Java applications can send and receive messages in a message queue provided by a Message Oriented Middleware. As message queues are based on first-in-first-out order, no reordering of the incoming message stream can take place. It is, however, possible to set a message selector that specifies which messages the client is interested in. Only these messages are then selected from the queue while the others are ignored. This mechanism allows simple queries on the header of JMS messages that are based on the conditional expressions of the SQL `WHERE` clause described above and use a similar syntax. Thus, a JMS message selector is also roughly equivalent to a single XVSMQL matchmaker function, while XVSMQL selectors and filter chains enable improved query capabilities compared to JMS.

Queries on structured data can also be found for XML documents, where XPath 1.0 [8] is a common language to extract information. An XPath expression consists of a sequence of location steps that contain an axis, a node test and a predicate. Starting from a given context node, the axis determines the navigation direction, e.g. direct children, ancestors or descendents can be examined. A node test specifies the name or type of nodes that are

included in the result set, whereas predicates compare certain attributes of a node similar to XVSMQL matchmakers. These predicates include comparisons, Boolean functions and simple arithmetic operators on any sub node, as specified by a path that may itself contain several location steps. For each location step of an XPath query, all nodes of the given axis that fulfill the node test and the predicate are selected and used as starting point for the next location step, until the result of the last location step is finally returned to the user. XVSMQL does not support the axis concept because queries are always evaluated on a single multiset or sequence of xtrees. Node tests and simple predicates that only use the child axis within their path and do not compare the values of multiple attributes, however, basically correspond to the capabilities of XVSMQL matchmaker functions. Moreover, the selector functions of XVSMQL have no equivalency in XPath, which may only return an unordered node set as a result.

Compared to Linda template matching [13], XVSMQL offers much more possibilities for querying data in the space. In Linda, it is not possible to retrieve tuples in a sorted way or to compare if a certain property is higher or lower than a specified value. This query mechanism only allows checking for exact equality of a field compared to a single value, while the matchmaker functions of XVSMQL enable more powerful comparisons. With XVSMQL, it is also possible to retrieve several entries at once, whereas Linda tuple spaces may only return one matching tuple per read operation. Template matching allows checking if a tuple has a specific cardinality. As XVSMQL permits access to properties via their position, the only way to test if an xtree has a particular count `n` of properties is to check that the property at path `n` exists and the one at path `n+1` does not. A simpler way would be to define an own matchmaker testing for the size of an xtree.

The analysis of the XVSM Query Language has shown that the presented approach is feasible because other well established query languages that operate on structured data use similar mechanisms to select data. Instead of using one of these existing query languages for queries on xtrees, the XVSM Query Language has been defined to retain control on the exact semantics of queries on the space. Other query mechanisms are not designed for the use in SBC middleware or offer insufficient expressive power like Linda template matching. Therefore, an own approach is used while useful features of other query languages are incorporated.

The XVSMQL definition is subject to change, as features that appear useful in the future might be included in the set of predefined query selectors and matchmakers. In the following, some possible extensions are listed:

- a matchmaker based on the size of a sequence or multiset xtree at a given path
- a variant of `cnt(n)` returning all, but at least `n` entries (otherwise the query fails)
- a variant of `cnt(n)` limiting the maximum number of returned entries to `n`

- allowing comparison between two or more different properties of an entry in a single property predicate, including simple arithmetic expressions for numerical values
- a matchmaker for string properties based on the SQL `LIKE` operator or on regular expressions in general

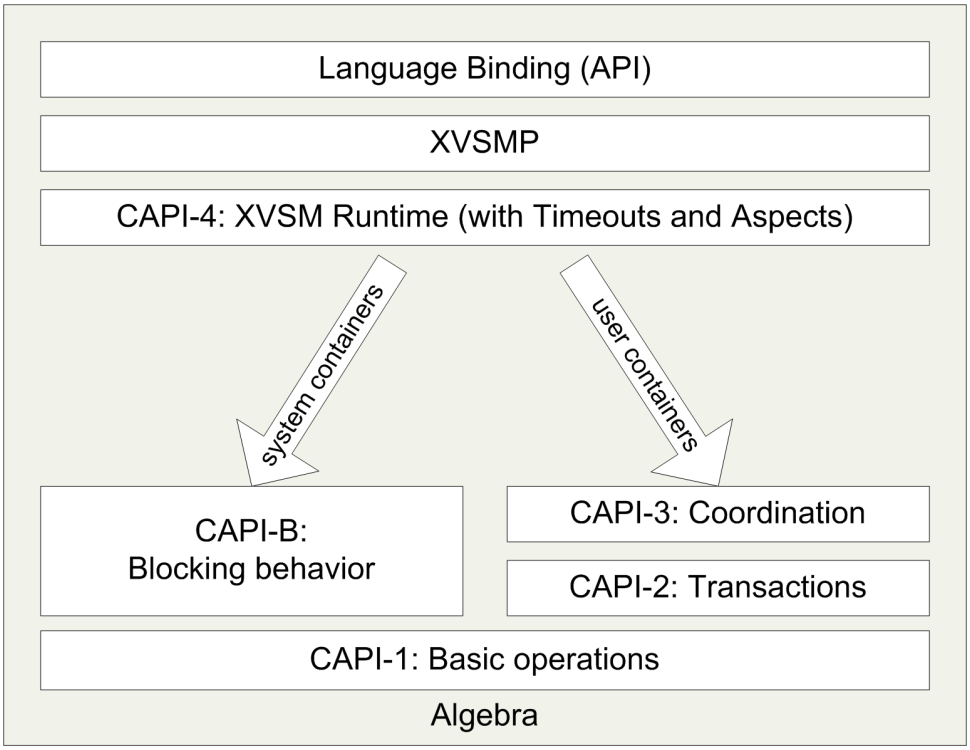

Figure 1: Layered architecture of XVSM

4 XVSM Core API

The semantics of the XVSM middleware are defined via a language-independent protocol called XVSMP, which is itself bootstrapped by several Core API layers. Each of these CAPI layers mainly uses functionality of the tier directly underneath, while every layer may use the algebraic foundation described in Sect. 2. Figure 1 outlines this architecture and shows its main elements, which are described in detail in sections 4.1 to 4.5.

The basic operations of CAPI-1 enable the creation of a simple space that can store all data necessary for higher CAPI layers to bootstrap themselves. This layer is a data storage that does neither support transactions nor any other form of coordination than using simple atomic write, read and take operations with explicit path specification and the query possibilities of XVSMQL. The higher CAPI layers use this storage provided by CAPI-1 for user data as well as for meta data that they need to fulfill their tasks. CAPI-2 adds transactional access to the space by aggregating several basic operations to a single atomic action, while CAPI-3 introduces the container concept with coordinators that enable data extraction mechanisms exceeding the query possibilities of XVSMQL. Since one single CAPI-3 operation consists of several accesses to the XVSM meta model and its effects must remain atomic, it relies on internal transactions provided by CAPI-2. The XVSM runtime model is specified with CAPI-4, which is responsible for the initialization of the meta model and for the scheduling and execution of user requests.

Furthermore, aspects that may enhance the semantics of CAPI-3 operations are defined at this level. The XVSMP layer maps the XML protocol to the internal xtree representation of requests and invokes the runtime in an asynchronous way. On top of the protocol, arbitrary language bindings can be built that send and receive XVSMP messages and thus can communicate with the space core.

The CAPI layers 1 to 3 are all synchronous and non-blocking, so they return immediately with a result after they have been called by an above layer, even if they currently cannot fulfill their task. This behavior guarantees that a CAPI-3 operation can never block the whole runtime by waiting possibly infinitely for some kind of event. Thus, the runtime machine keeps complete control over the chronology of its requests.

To bootstrap the scheduling of requests with CAPI-4, the non-blocking CAPI layers are not sufficient. Therefore, a variant of CAPI-1 called CAPI-B is introduced, which extends the basic operation set with additional functions that may block until they are successful. CAPI-B is used by the runtime machine to access special runtime containers that are part of the XVSM meta model and temporarily store active requests and supplementary information. For the execution of the actual requests on user containers, the runtime machine uses the non-blocking CAPI-3.

All CAPI operations return an xtree containing a status code together with the actual result if applicable. The possible status codes are defined as follows:

- `OK`: Operation successfully completed.
- `DELAYABLE` (only CAPI-1, 2 and 3): The operation cannot be executed at the moment, but it should be retried in the future. This happens, for example, if a read operation tries to read one entry from an empty container.
- `LOCKED` (only CAPI-2 and 3): One or more data structures needed for the execution are locked by another transaction. The operation should be retried when the locks are released.
- `NOTOK`: The operation cannot succeed and should raise an error to the caller.

If the status is not `OK`, a status information property indicating the reason why the operation is not able to finish is given instead of the result. Functions of CAPI-2 and 3 check the result status of any invoked lower level CAPI method. If an error message is received, the function usually also returns immediately with the obtained error status.

In contrast to the lower layers, CAPI-4 is able to block until a request can be fulfilled within a given timeout. As long as the runtime machine receives a `DELAYABLE` or `LOCKED` result from a call to the corresponding CAPI-3 operation, the request can be rescheduled and thus has the chance to be fulfilled at a later time. When a timeout or a result

with status `OK` or `NOTOK` occurs, an answer entry is generated for the protocol layer that includes the request result or error message, respectively.

4.1 CAPI-1: Basic Operations

All functionality of the XVSM core is based on the simple data access mechanisms of CAPI-1. It is possible to write specified xtrees into the space at a certain position and to read and take xtrees — fulfilling a given XVSMQL query — from the space. Thus, the access possibilities at this lowest level already resemble the write, read and take operations of the user API, enabling a clean bootstrapping process of the core functionality with a minimum of basic operations. At this layer, user and meta data are not distinguished, so it is not only possible to read entries, but also the content of whole containers or of a certain meta property at a well-known path. It must be noted that the available operation set does not include an operation to update a property at a given path, but this can be easily simulated by a take followed by a write operation on the same path. Of course, it would be possible to define separate data manipulation operations to get, set, update and delete properties of the meta model that have a fixed path and need not be queried by XVSMQL. However, to keep the model simple such methods are omitted in favor of the supported operations `write1`, `writeBulk1`, `writeL1`, `read1` and `take1`. With `write1` and `writeBulk1`, one or more xtrees, respectively, are written at a certain path of the space with automatically generated labels that are also returned in the result, while `writeL1` uses a specifically declared label, which is useful for writing meta properties located at a fixed position within the meta model. The `read1` and `take1` operations support XVSMQL queries on a specified sub xtree of the space.

CAPI-1 operations are considered to run as atomic operations, so at any time only one CAPI-1 operation may access the same data structure. As mentioned in Sect. 4, all CAPI-1 operations return an xtree containing a status code and potentially also a result or a status information property in case of errors. Table 2 shows the arguments and return values of all CAPI-1 operations with the types of all parameters in brackets. In the following, the exact semantics of the individual functions are specified, broken up into the two categories of write and query operations, accompanied by a simple implementation in pseudo code syntax, using the algebraic facilities defined in Sect. 2.

	Arguments	Result
writeL1	Root (Sequence/Multiset xtree), Path (String), Label (String), Value (XTree)	-
write1	Root (Sequence/Multiset xtree), Path (String), Value (Xtree)	Label (String)
writeBulk1	Root (Sequence/Multiset xtree), Path (String), Values ([Xtree])	Labels ([String])
read1	Root (Sequence/Multiset xtree), Path (String), Query (String)	Query result (Sequence/ Multiset xtree)
take1	Root (Sequence/Multiset xtree), Path (String), Query (String)	Query result (Sequence/ Multiset xtree)

Table 2: CAPI-1 operations

Basic write operations

```
1 writeL1(xt, path, lbl, val)
2   parent := xt.path
3   if (parent = undef)
4     return [statuscode:"NOTOK", info:"InvalidPath"]
5   else
6     parent := parent ⊔ [lbl:val]
7     return [statuscode:"OK"]
```

Listing 1: Definition of writeL1

The three write operations of CAPI-1 are defined in a very similar manner. As an example, Listing 1 shows the definition of `writeL1` for parent xtrees that are multisets, whereas the definition for sequence xtrees is analogous. Starting from the root xtree `xt`, this operation tries to write a property with specified label `lbl` and xtree value `val` into a child xtree of the root specified by the `path` argument. The path may not include wildcards, but can be ambiguous due to duplicate labels. In this case, only one xtree is chosen indeterministically for the write operation. If no xtree at the given path exists, an error is indicated. The write operation itself is realized via a multiset union (or a sequence concatenation, respectively) on the target xtree. In contrast to `writeL1`, the `write1` function does not require a label as a parameter. Instead, an automatically created label is used when writing the value xtree to the space. To enable higher-level CAPIs to access the written xtree after the operation completes, the label is returned in the result xtree. This label is unique within its parent xtree to avoid indeterministic behavior when accessing the property via its path. The `writeBulk1` operation is defined like `write1` with the difference that a list of value xtrees is given as an argument instead of just a single value. All of these xtrees are written to the given path with separate unique labels, which are also returned as a list in the result xtree, using the same order for the labels as for their corresponding value xtrees in the argument list.

Basic query operations

```
1  take1(xt, path, query)
2    parents := {x | x = xt.path}
3    if (query = ϵ)
4      forall p ∈ parents
5        xt := removeXtree(xt, p)
6      resultXtree := buildResultXtree(parents)
7      return [statuscode:"OK", result:resultXtree]
8    else
9      searchXT := createSearchXtree(parents)
10     (entries, errXT) := applyQuery(searchXT | query)
11     if (errXT = [])
12       forall e ∈ entries
13         xt := removeXtree(xt, e)
14       resultXtree := buildResultXtree(entries)
15       return [statuscode:"OK", result:resultXtree]
16     else
17       return [statuscode:"DELAYABLE", info:errXT]
```

Listing 2: Definition of take1

The query operations `read1` and `take1` return xtrees in a specified area that match a given XVSMQL query. Both return equal results when used with the same parameters. The only difference is that `take1` has the additional side effect of removing the found xtrees from the space, therefore the implementation of `take1` shown in Listing 2 can be easily adapted for `read1`. In contrast to the write operations, wildcards within paths are possible to extend the search area. In line 2, all xtrees of the root `xt` with matching path are stored in the `parents` variable. Two query modes are supported: If an empty query is specified, the xtrees in `parents` are directly returned. For `take1`, these xtrees are removed from their own parent xtrees in this case with the `removeXtree` function. This mode is useful when accessing a property directly by its full path similar to the previously defined `writeL1`. It must be noted, however, that there is no guarantee that the result contains exactly one xtree value, as there could be any number of xtrees at the given path or none at all. In all cases the query operation returns with an `OK` status. If the specified query is non-empty, it is applied to a single search xtree which is generated from the contents of the xtrees in the `parents` variable (line 9). Properties of multiset xtrees are put into a single multiset, while properties of sequence xtrees are put into a single sequence if the parent xtrees themselves have an unambiguous order. If sequence and multiset xtrees are mixed or if the order of several sequence xtrees is not defined (e.g. when they are located in the same parent multiset xtree), the structure of the resulting search xtree remains unspecified. Other xtrees in the `parents` variable that are not sequences or multisets (like simple values) do not influence the search xtree as they do not contain any child xtrees. For retrieving all entries of this search xtree, a query consisting only of the `id` selector can be used. The `applyQuery` function invokes the XVSMQL query on the search xtree and returns either the result entries or an error. In the latter case, the query operation

returns with a `DELAYABLE` status together with the error message. Otherwise, an `OK` status is returned together with the result and, for `take1`, the matching entries are also removed from their respective parent xtrees. For both modes, the result entries are prepared by the `buildResultXtree` function before the operation returns, as the property consisting of label and value is not sufficient to identify the location of a found entry because the path may contain wildcards that need to be resolved. This is done by enriching the result entries with the declaration of their complete path within the root xtree. The result of a `read1` or `take1` operation is therefore a sequence or multiset of entries like in the following example: $\langle$[`path`:"a/b/c", `value`:E_1], [`path`:"a/d/c", `value`:E_2]$\rangle$

4.2 CAPI-2: Transactions

The basic operations specified in the previous section are defined to be atomic but the transactional safety of an action consisting of several CAPI-1 operations cannot be guaranteed. Therefore, it becomes necessary to group these actions into transactions.

4.2.1 Transaction model

In database theory, transactions are usually described with the so-called ACID properties [15]. This definition, which is an acronym for the terms *atomicity*, *consistency*, *isolation* and *durability*, can also be applied to spaces [47]:

- **Atomicity:** Either all operations of a transaction occur in an atomic way or none of them do.
- **Consistency:** The space must be in a consistent state after the transaction completes.
- **Isolation:** Concurrent transactions should not affect each other.
- **Durability:** Changes made by transactions must be permanent.

CAPI-2 ensures these properties by using a pessimistic locking model on the xtrees of the space, although durability can only be guaranteed as long as the core is running. Each CAPI-2 operation tries to lock the data structures it accesses with adequate locks. If a lock cannot be acquired because an incompatible lock of another transaction already exists, the whole operation returns with status `LOCKED`. Although both status values `LOCKED` and `DELAYABLE` indicate that the corresponding request needs to be rescheduled, they are distinguished because the runtime handles these types of requests slightly differently (see Sect. 4.4). The process of acquiring a lock has to occur atomically, so that two parallel CAPI-2 operations may not set an exclusive lock on a previously unlocked entity in a concurrent way.

The influence of concurrent transactions on each other cannot be completely eliminated if they try to invoke conflicting operations on the same data structure, as only one transaction may succeed while the others must wait until this transaction completes. In some cases, it is beneficial to use a strict isolation policy that forbids changes or even read access by other transactions until commit or rollback, while in other cases, a maximum of concurrency is sought at the expense of a clean isolation between transactions. Therefore, the ANSI/ISO SQL standard defines four different isolation levels for databases [20]:

- **Read uncommitted:** Uncommitted changes can be seen by other transactions, so *dirty reads* are possible.
- **Read committed:** Only committed data can be read by other transactions, but there is no guarantee that the records retrieved within a transaction are still available when it commits, thus allowing *non-repeatable reads*. Data changed within a transaction cannot be accessed by other transactions until the first transaction commits or rollbacks, which is achieved with write locks.
- **Repeatable read:** In addition to the exclusive write locks, also read locks are introduced which guarantee that read values remain valid for the whole transaction. It is, however, still possible within a transaction that consecutive queries with the same parameters return different results, as additional values added by other transactions could influence the query (*phantom reads*).
- **Serializable:** This strictest isolation level avoids the previously mentioned isolation problems by letting each transaction work on a completely isolated view of the data. The effects of the transactions must appear as if they are executed in some sequential order.

Spaces have slightly different isolation requirements than databases, as no update operation is supported. On the other hand, the blocking behavior of space operations may lead to prolonged transactions. For spaces, the isolation level "serializable" is too strict because blocking read and take operations within a transaction need to wait for data written by other transactions to allow a meaningful coordination. Therefore a complete isolation of transactions is not feasible. With "read uncommitted", no isolation between transactions is ensured, which is usually not viable for the coordination of concurrent processes, as one peer may see inconsistent data that is currently processed by another transaction. However, this mode enables the best performance when transactional integrity is not required. The isolation level "read committed" offers more practical semantics for the use in a space if read operations are still allowed to read entries that are currently being taken by another open transaction. This behavior can be useful for a value that

often needs to be updated, which is modeled with a take and a write operation within a transaction. In this case, a concurrent read always retrieves a matching value, either the old entry or the new one, while a transaction using "repeatable read" would not allow the read operation until the transaction invoking the update finishes. "Read committed" lacks, however, the possibility to prevent access to an entity by applying a lock on it. As a result, the default transaction model of CAPI-2 supports "repeatable read", which also corresponds to the isolation level used in JavaSpaces [47]. Due to the query semantics of XVSM, however, read and take operations are not really repeatable as the xtrees are not referenced directly but only by a combination of a possibly ambiguous path and a query chain. The transaction model only guarantees that the results of the first query are still available when the second query occurs, but phantom reads might lead to a different result for the second query if new entries are written to the space that affect the outcome of the XQ. In the future, modes for "read committed" and "read uncommitted" may also be added to extend the possible semantics of XVSM operations.

In XVSM, two kinds of transactions are distinguished: User transactions can be created and committed or rollbacked via the user API, while sub transactions are needed for internal use within the runtime to encapsulate a single operation during execution. A sub transaction is linked to a user transaction and has access to all of its locked data, but a sub transaction can also acquire its own locks. Figure 2 depicts the relation between user and sub transactions. A runtime operation, which corresponds to the execution of a user request by the runtime, uses a single sub transaction for its calls to CAPI-3. Due to aspects, one runtime operation may consist of multiple CAPI-3 operations (see Sect. 4.4). The CAPI-3 methods themselves call several CAPI-2 operations to realize their behavior. Within a user transaction, several runtime operations and thus sub transactions can be issued. If a runtime operation fails or must be rescheduled, all changes to the space that have been made by this operation must be revoked, so the sub transaction is rollbacked by the runtime. The surrounding user transaction, however, is not affected by this partial rollback of its changes and remains active. If the request succeeds, the sub transaction commits and transfers all of its locks to its parent user transaction. Multiple sub transactions of the same user transaction may be executed concurrently, but one sub transaction may not access data that is locked by another one until it finishes and transfers the lock to the user transaction, thus making the change visible. Sub transactions must treat locks of other active sub transactions of the same parent user transaction like locks of foreign user transactions because otherwise they could read changes of a concurrent operation that is later rollbacked. The transactional CAPI-2 operations need to be called with parameters for the user transaction and the sub transaction; the latter one is provided by the runtime to encapsulate a single request execution.

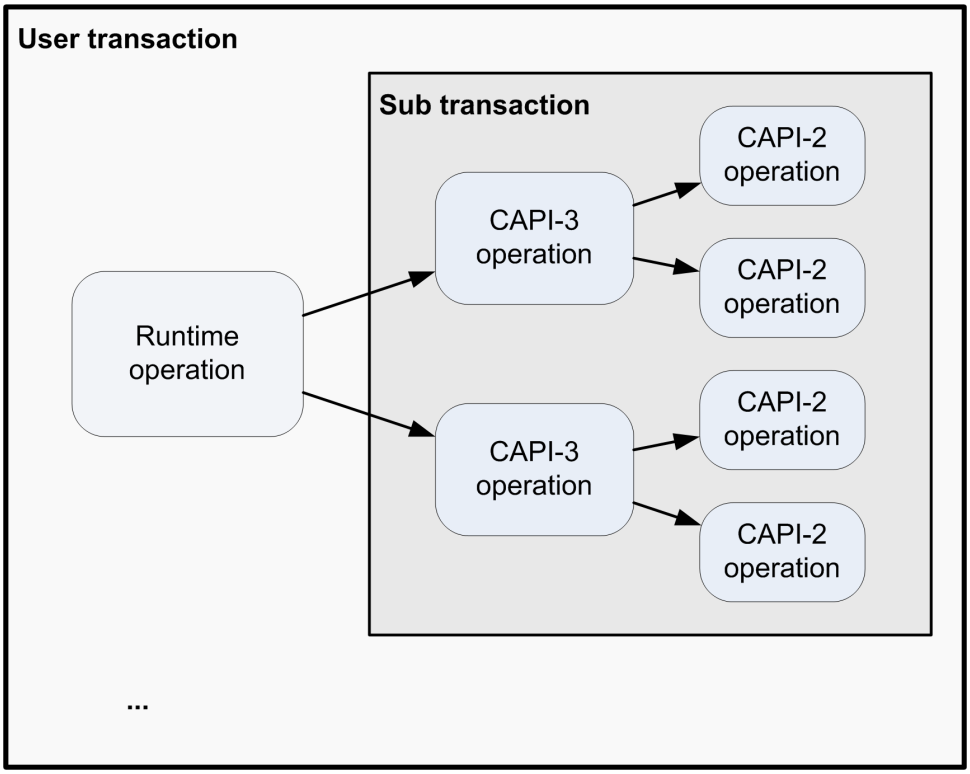

Figure 2: User and sub transactions

4.2.2 Locking semantics

There are three main lock types in the model, corresponding to the three basic operations:

- **insert lock:** These xtrees were newly created and should be invisible for other transactions.
- **delete lock:** These xtrees will be deleted, so other transactions that try to access them should wait until the transaction that holds the locks rollbacks or commits. This is an exclusive lock, so locks of other transactions must not exist on xtrees with delete locks.
- **read lock:** These xtrees are read by one or more transactions and must not be deleted. An xtree can have read locks of multiple transactions, so this is a shared lock.

A take operation does not remove the xtrees from the space immediately, it just marks them as deleted via the delete lock. Read locks can be upgraded to delete locks if no other reading transaction is present. If an xtree has both an insert lock and a delete lock, the xtree is invisible to other transactions, as it only temporarily exists within one transaction. An exclusive read lock can also be set manually on an xtree by invoking a special CAPI-2 method. This lock has the same meaning as a delete lock when checking

Lock type	Lock position	write X	take X	read X
insert lock	on parent of X	-	-	-
	on X	-	not visible	not visible
	on child of X	-	locked	not visible
delete lock	on parent of X	locked	locked	locked
	on X	-	locked	locked
	on child of X	-	locked	locked
read lock	on parent of X	OK	locked	OK
	on X	-	locked	OK
	on child of X	-	locked	OK

Table 3: Locking compatibility for locks of foreign transactions

lock compatibility but the xtree will not be deleted from the space. On commit, all locks of a transaction are removed along with all xtrees marked as deleted. On rollback, the same is done except that all xtrees with an insert lock are removed instead of the deleted ones.

In contrast to database tables, data in an XVSM space are not stored in flat tuples but in hierarchic xtrees. Therefore the transaction model has to consider locks on parent and child xtrees of an accessed entity as well. A write operation is only possible if no parent xtree is locked for delete by any other transaction. Similarly, a take operation is not allowed if any parent or child xtree has a foreign read or delete lock. Also, no child xtree may have a write lock. An xtree can also only be read when no parent or child xtree are exclusively locked by another transaction. However, not only direct children and the parent of an accessed xtree have to be examined but also all of its ancestors and descendants by recursively following the parent and the children, respectively, of the xtree. These indirect parents and children share the same locking semantics as direct parent and child xtrees. The compatibility of locks with operations of a foreign transaction are also shown in Table 3. An entry of "not visible" in the table means that the operation may succeed but the locked xtrees are ignored and not included in the result. If no entry exists, the combination of lock type and operation is not possible, e.g. a child xtree of an xtree that is just being written to the space cannot be locked yet by another transaction because only the own transaction is able to see it.

This pessimistic locking strategy is a variant of strict Two-Phase locking (2PL) protocol [2] because it uses exclusive locks (for take) and shared locks (for read). Xtrees with insert locks (for write) can usually be ignored because they are invisible to other transactions until commit. During every CAPI-2 operation, locks are acquired, whereas these locks are released when a sub transaction fails to acquire all needed locks during its execution and the corresponding request needs to be rescheduled. At user transaction commit or rollback, all locks set by all of the transaction's operations are released at

once. Due to the pessimistic approach, a commit must never fail, as conflicts are already recognized beforehand. The main difference to classic 2PL is that a runtime operation is not blocked when it finds a non-compatible lock. Instead it is rollbacked and rescheduled by the runtime machine, so a classic deadlock cannot occur. It is, however, possible to get stuck in a livelock when two requests from different transactions are getting rescheduled constantly due to locks held by the other transaction. In this case, one transaction must be aborted by the runtime.

In the meta model, all locks are stored as lock entries in a special container `_locks`. They contain the lock type (`READ`, `EXCL`, `DELETE` or `WRITE`), the complete path to the locked xtree, the transaction id and the operation id that identifies the sub transaction, as shown in the following example:

[`type`:"READ", `path`:"containers/e1/entries/e3", `tx`:12, `op`:1]

When the sub transaction commits, the operation id is removed from the lock, indicating that the lock is now valid for all sub transactions of the specified transaction. It is necessary that each locked xtree has a unique path, otherwise it cannot be determined which entity should actually be locked. Therefore, CAPI-2 offers no equivalency for the `writeL1` operation which is used for updating meta properties. When such a property is taken from the space with CAPI-2, the old xtree still exists in the space until the transaction commits, even though it is no longer accessible via the own or foreign transactions. If a write operation would write a new value at the same path, there would be two lock entries for the same path, one with a delete lock and one with an insert lock, and it would be impossible to determine which of the two xtrees is the new one.

The concept of a central lock container requires that every CAPI-2 operation needs exclusive access to the container for a short time while it tries to acquire its locks. It eases, however, the management of locks in the meta model. If the locks were attached directly to the locked entities, it would be necessary to recursively lock the whole hierarchy from the root xtree to the xtree that needs to be accessed. With the `_locks` container, the parents and children need not be explicitly locked. When acquiring a lock, incompatible locks on parent and child xtrees are simply identified by comparing the paths. In an actual XVSM implementation, more sophisticated strategies for applying locks can be used, but for the specification of the formal model this simple centralized lock management is sufficient.

4.2.3 Transaction logging

To enable a simple commit or rollback of a (sub) transaction, every locking action is logged in the meta transaction container `_txC`. This structure contains an entry for every

transaction, identified via a label that represents its id. This entry consists of a log for each sub transaction and a transaction status property, which can be any of the values `RUNNING`, `COMMITTING`, `COMMITTED`, `ABORTING` and `ABORTED`. The sub transaction log, which is labeled with its operation id, contains a property that specifies the type of the operation, a status property (`ACTIVE`, `FINISHED` or `CANCELLED`) and a log listing all locking actions. The log entries contain a type and a path property and simplify the commit or rollback process. Each set lock in the `_locks` container is referenced directly with type `LOCK-INSERTED`, while the path of inserted or deleted properties is listed with `PROP-INSERTED` and `PROP-DELETED` entries, respectively. A simple example transaction entry in the `_txC` container might look as follows:

```
tx9 : [status:"RUNNING",
      log : [op1 : [capiName:"take",
                    status:"CANCELLED",
                    operationLog : ⟨
                      [type:"LOCK-INSERTED", path:"_locks/e1"],
                      [type:"LOCK-INSERTED", path:"_locks/e2"],
                      [type:"PROP-DELETED", path:"containers/e1/entries/e3"]
                    ⟩],
             op2 : [capiName:"write",
                    status:"FINISHED",
                    operationLog : ⟨
                      [type:"LOCK-INSERTED", path:"_locks/e3"],
                      [type:"PROP-INSERTED", path:"containers/e1/entries/e4"]
                    ⟩]]]
```

4.2.4 Transactional operations

Similar to CAPI-1, all CAPI-2 methods return an xtree containing one of the status codes `OK`, `NOTOK`, `LOCKED` and `DELAYABLE`, as well as the actual result. The function parameters and their types are listed in Table 4. All transactional methods require transaction and operation parameters, which must at first be initialized via `createTransaction2` and `startOperation2`. Internally, CAPI-1 operations are used to manipulate the `_txC` and `_locks` meta containers and to invoke the actual data access. In the following, the exact semantics of the CAPI-2 operations are specified.

	Arguments	**Result**
createTransaction2	Space (Space)	Transaction Id (String)
commitTransaction2	Space (Space), Transaction Id (String)	-
rollbackTransaction2	Space (Space), Transaction Id (String)	-
startOperation2	Space (Space), Transaction Id (String), Operation name (String)	Operation Id (String)
finishOperation2	Space (Space), Transaction Id (String), Operation Id (String)	-
cancelOperation2	Space (Space), Transaction Id (String), Operation Id (String)	-
write2	Space (Space), Path (String), Value (Xtree), Transaction Id (String), Operation Id (String)	Label (String)
writeBulk2	Space (Space), Path (String), Values ([Xtree]), Transaction Id (String), Operation Id (String)	Labels ([String])
read2	Space (Space), Path (String), Query (String), Transaction Id (String), Operation Id (String)	Query result (Sequence/ Multiset xtree)
take2	Space (Space), Path (String), Query (String), Transaction Id (String), Operation Id (String)	Query result (Sequence/ Multiset xtree)
setExclusiveLock2	Space (Space), Path (String), Transaction Id (String), Operation Id (String)	-
getReadable2	Space (Space), Path (String), Transaction Id (String), Operation Id (String)	Result paths ([String])
getTakeable2	Space (Space), Path (String), Transaction Id (String), Operation Id (String)	Result paths ([String])

Table 4: CAPI-2 operations

Transaction management operations: The `createTransaction2` operation generates and returns a unique transaction id while it initializes a new transaction entry with status `RUNNING` at the path `_txC/`*`id`*. A new sub transaction is created with the `startOperation2` command for a transaction `tx`, which writes an empty operation log entry with status `ACTIVE` into the meta model at position `_txC/`*`tx`*`/log/`*`opid`*, where `opid` is a unique identifier within the transaction. A sub transaction can be committed via `finishOperation2`, which removes the operation id properties from all locks listed in the operation log corresponding to the specified transaction and operation id and thus transfers the locks to the parent transaction. It also sets the operation status to `FINISHED`. A rollback on a sub transaction is invoked with `cancelOperation2`, setting the operation status to `CANCELLED`. This operation removes all locks that are found via the operation log and also takes all properties from the space that are referenced by a `PROP-INSERTED` entry. When `commitTransaction2` or `rollbackTransaction2` are invoked, the transaction status is immediately set to `COMMITTING` or `ABORTING`, respectively, which prevents new operations from being started, while currently active ones are still allowed to finish before the transaction ends. All entries within operation logs with status `FINISHED` are then processed: Locks referenced via `LOCK-INSERTED` entries are removed in both cases. For `commitTransaction2`, properties that are registered in `PROP-DELETED` entries are definitely deleted from the space, whereas for `rollbackTransaction`, newly created properties referenced with `PROP-INSERTED` in the log are removed. Finally, the transaction status is set to `COMMITTED` or `ABORTED`, respectively, and the log entry can be deleted if needed.

Transactional space operations: The operations `write2`, `writeBulk2`, `read2` and `take2` have the same functionality as their CAPI-1 equivalents, while they additionally try to acquire appropriate locks on the data they access. If a conflicting lock is found, the operation returns with status `LOCKED`. Additionally, the returned xtree contains a `lockType` property that indicates if the sub transaction that holds the lock has already committed or if it is still active, which is necessary for the scheduling semantics of the runtime (see Sect. 4.4.1). The lock types can be distinguished by looking at the `op` property of the lock entry. If it exists, the associated sub transaction is still active, otherwise it has already transfered its locks to the parent user transaction.

The `write2` and `writeBulk2` operations must check at first if the requested path is visible for the given sub transaction, meaning that no foreign write lock or own delete lock must be active on any parent xtree of the entries that should be written, otherwise status `NOTOK` is returned. Then, the operation must verify that no foreign delete lock is active on any parent xtree, which would yield a result of status `LOCKED`. If these preconditions are fulfilled, the specified entries are written to the space together with

write lock entries in the `_locks` container as well as `LOCK-INSERTED` and `PROP-INSERTED` entries in the operation log. When invoking `read2` or `take2`, a copy of the search path is generated where all invisible xtrees with foreign write locks or own delete locks are removed. The copy is needed because a CAPI-1 query that uses the space directly would include the invisible xtrees and thus distort the result. Therefore, a CAPI-1 read operation is issued on the modified copy and the resulting entries are locked for read or delete, respectively, with corresponding `_locks` container entries, if no conflicting locks appear. Additionally, for each lock a `LOCK-INSERTED` entry is added to the operation log, as well as `PROP-DELETED` entries in the case of `take2`. A read operation is valid if neither the found entries nor their ancestors or descendants are locked by a foreign delete lock, while for a take operation also foreign insert or read locks are disallowed. Otherwise status `LOCKED` has to be returned.

Explicit locking operations: The `setExclusiveLock2` method locks an xtree for exclusive access and thus guarantees that no other transaction may access it until commit or rollback. This function enables a transaction to exclusively access an xtree without the interference of any other transactions because only the own transaction may read or take the xtree or any of its direct or indirect child xtrees. Also, other transactions may not write anything into the xtree. This behavior can be useful to ensure that only one transaction modifies a certain data structure at once. In contrast to `read2` and `take2`, no query can be specified, so the exact path of the xtree to lock must be known. The locking semantics of this operation are very similar to those of `take2` with the difference that no `PROP-DELETED` entry is added to the log and that an exclusive lock is used instead of a delete lock. In contrast to xtrees locked for delete, xtrees with exclusive locks are still visible by the own transaction. Therefore, these xtrees are not removed from the space on commit, but otherwise the two lock types are the same. With `getReadable2` and `getTakeable2`, all properties of an xtree at the specified path that could be locked successfully for read or take, respectively, are determined. These functions return a list of the paths of all these entries without actually locking the corresponding xtrees. This behavior is useful for indeterministic coordinators that try to preferably pick unlocked entries. All three explicit locking operations allow wildcards in their path argument, which enables them to access multiple xtrees at once.

4.3 CAPI-3: Coordination

While CAPI-1 and 2 control the access on arbitrary xtrees within the XVSM meta model, the focus of CAPI-3 lies on user containers and their entries. Therefore, the operations of this layer resemble the functionality of the user API with the exception of aspects and timeouts that are introduced in CAPI-4. A container in CAPI-3 consists of a name,

user data in the form of entries and several coordinators that manage these entries. Entries are xtrees of arbitrary structure although some coordinators may require a special entry structure. A coordinator is an exchangeable module consisting of several functions that control how entries are written to and retrieved from a container. There exist several predefined coordinators for FIFO queues, Linda template matching, access via keys and other coordination forms but arbitrary modules can be added to the space that support more sophisticated patterns. A coordinator must define so-called *accountant functions* that are invoked whenever an entry is written, read, taken or deleted within a container. For each write operation, the corresponding coordinators can update their internal meta data which reflect how the entry can be retrieved again via the coordinators' *query functions*. These meta data are stored in *coordination xtrees* (CoXT), which exist for every coordinator of a container. Coordinators may also directly attach meta data to entries by writing into their so-called *entry CoXTs*. A coordinator can also decide if an operation on a specific entry should be allowed or if a coordinator-specific constraint is violated so that it must fail. The capabilities of accountant functions are, however, limited. A coordinator only has full access to its own private container CoXT as well as to its own entry CoXT for each user entry. These entry CoXTs, however, may not be changed after the entry is initially inserted into the container. Additionally, the data of user entries may be read but not changed, while the CoXTs of other coordinators or further locations of the space may not be accessed at all. The `write3`, `read3`, `take3` and `delete3` operations of CAPI-3 are responsible for calling the accountant functions of all registered coordinators in the correct order. Thus, the coordinators are running in a sandbox that only allows limited access to the space, which prevents them from interfering with each other and makes them easily exchangeable. To realize the described functionality, all CAPI-3 operations use the transactional CAPI-2 methods. The coordinator's accountant and query functions also have access to CAPI-2 with the restrictions mentioned above.

When a new container is created, the user may specify several coordinators that manage the entries. A coordinator can either be declared as *obligatory* or *optional* for this container. The accountant functions of an obligatory coordinator must be called for every operation on this container, so that such a coordinator always has a complete view of all entries. An optional coordinator, however, only manages entries that are explicitly written with this coordinator, while other entries of the container remain invisible. Every coordinator may have an arbitrary number of arguments that are used for correct initialization of the CoXT. Some coordinators also require additional information whenever an entry needs to be written using this coordinator. In this case, the arguments are encapsulated in a so-called *write selector*. When the user wants to read, take or delete entries from the container using the coordination mechanism of a specific coordinator, a *read selector* is used to pass the arguments. An additional way to pass information to

the coordinator functions is via the context xtree, which contains meta data that can be subsequently enriched with arbitrary properties by the user, the runtime and the CAPI-3 operation that calls the coordinator. This context could, for instance, include information from the runtime about the current time or the name of the user invoking a request. Some coordinators may rely on particular well-defined context properties to be able to work properly, whereas others do not access the context xtree at all. Coordinators of the same type can be defined more than once for a container as long as they are distinguished via a unique id that is specified when the container is created. For example, it could be feasible to define two optional FIFO coordinators which are alternately invoked on written entries, depending on their meaning. Thus, entries can be retrieved via two independent queues. E.g., in a container of book entries, one queue can be used as a reading list for user A, whereas another queue represents the reading list of user B. Every container has an implicit obligatory coordinator called `SystemCoordinator`, which guarantees that a container never exceeds its bounds of maximum allowed entries, as specified at creation time of the container. This coordinator also allows simple random access on available entries via read and take queries.

4.3.1 Coordinator interface

As usual, the operations of a coordinator return an xtree containing the status code and potential errors. The parameters of these methods and their types are shown in Table 5. The methods have access to the context xtree of the enclosing CAPI-3 operation, which may include relevant meta data, and to the transaction and operation ids that are necessary to ensure the transactional integrity of the operation.

Init function: When a new container is created, this method initializes the internal data structures of the coordinator using the available parameters. This information is then returned in the form of a container CoXT that can later be used by the accountant and query functions.

Accountant functions: Each of the four accountant functions `onInsert`, `onRead`, `onRemove` and `onDataReturn` is called on single entries of a container. To decide if an operation is allowed by the coordinator so that status `OK` should be returned, these functions may access the container CoXT located at *`containerPath`*`/coxts/`*`coordinatorID`* and the entry CoXT of the coordinator included in the specified entry. The entry argument is given as a property instead of an xtree. Thus, the entry's label is also available to allow its use as a reference in the internal lists and indices of the coordinator. The `onInsert` accountant function is used whenever a new entry is written to the container. The write selector indicates how the coordinator should store the entry, e.g. a vector coordinator

	Arguments	**Result**
init	Coordinator arguments (Xtree), Context (Xtree)	Container CoXT (Xtree)
onInsert	Space (Space), Container path (String), Entry (Property), Write selector (Xtree), Context (Xtree), Transaction Id (String), Operation Id (String)	Entry CoXT (Xtree)
onRead	Space (Space), Container path (String), Entry (Property), Context (Xtree), Transaction Id (String), Operation Id (String)	-
onRemove	Space (Space), Container path (String), Entry (Property), Context (Xtree), Transaction Id (String), Operation Id (String)	-
onDataReturn	Space (Space), Container path (String), Entry (Property), Context (Xtree), Transaction Id (String), Operation Id (String)	Meta properties (Xtree)
query	Space (Space), Container path (String), Read selector (Xtree), inEntries (XTree), Context (Xtree), Transaction Id (String), Operation Id (String)	outEntries (Xtree)

Table 5: Coordinator functions

requires a position to be specified. The method may return an entry CoXT that has to be added to the entry by the write operation for later uses of this coordinator. The `onRead` method, which is called for every read or taken entry, is usually only useful for special coordinators that depend on the order in which entries are accessed within a container, like a coordinator returning entries that were least recently used. With the `onRemove` accountant function, the coordinator has the possibility to clear a removed entry from its internal data structures, while the `onDataReturn` method returns special meta properties of the coordinator that should be attached to the entry before it is returned to the user, like, e.g., the position of an entry using vector coordination. In contrast to the other accountant functions, the `onDataReturn` method may not change the CoXT.

Query function: Analogous to the query mechanism of XVSMQL, the coordinator query functions can be invoked in a chain. Each query function may take an arbitrary number of arguments given in the read selector parameter as well as a list of incoming entries that were selected by the previous coordinator. For the first coordinator, all entries of the container are eligible, but further coordinators may only select and return entries that are listed in their `inEntries` argument. Query functions may only read the entry data as well as the entry and container CoXTs but they are not allowed to change any of the CoXTs, as this is the task of the accountant functions which are called on the

	Arguments	Result
createContainer3	Space (Space), Container name (String), Size (Integer), Obligatory Coords ([(Coord, String, Xtree)]), Optional Coords ([(Coord, String, Xtree)]), Context (Xtree), Transaction Id (String), Operation Id (String)	ContainerRef (String)
destroyContainer3	Space (Space), ContainerRef (String), Context (Xtree), Transaction Id (String), Operation Id (String)	-
lookupContainer3	Space (Space), Container name (String), Context (Xtree), Transaction Id (String), Operation Id (String)	ContainerRef (String)
setContainerLock3	Space (Space), ContainerRef (String), Context (Xtree), Transaction Id (String), Operation Id (String)	-
write3	Space (Space), ContainerRef (String), Entries with write selectors ([(XTree, [(String, Xtree)])]), Context (Xtree), Transaction Id (String), Operation Id (String)	-
take3	Space (Space), ContainerRef (String), Read selectors ([(String, Xtree)]), Context (Xtree), Transaction Id (String), Operation Id (String)	Result entries (Xtree)
read3	Space (Space), ContainerRef (String), Read selectors ([(String, Xtree)]), Context (Xtree), Transaction Id (String), Operation Id (String)	Result entries (Xtree)
delete3	Space (Space), ContainerRef (String), Read selectors ([(String, Xtree)]), Context (Xtree), Transaction Id (String), Operation Id (String)	-

Table 6: CAPI-3 operations

final result entries. Only entries that are managed by the corresponding coordinator are visible for a query function. Therefore, entries that are not registered for the coordinator are filtered out before the actual selection criteria are applied.

4.3.2 Container operations

Table 6 shows the operations of CAPI-3, which are called directly from the runtime to execute user requests on the space and return the same status codes as CAPI-2. Beside the transactional parameters, all of these methods also have access to the runtime context of the respective operation although not every CAPI-3 operation actually uses this context.

createContainer3: This method creates a new container with specified name and size by initializing the container meta structure in the `containers` xtree of the space. The automatically generated label of the newly created container within this structure can be used as a unique reference and is therefore returned to the caller. If no name is given, the container remains anonymous so that it cannot be found via `lookupContainer3`. In this case, the only way of accessing it is by using the returned container reference. The size of the container depicts the maximal number of entries that can be stored in it, which is ensured by the `SystemCoordinator` that is initialized with the given bounds parameter. If the constant value `INFINITE` is used, the container remains unbounded. The obligatory and optional coordinators are given as list of triples that include the coordinator, its unique id within the container and the arguments for the coordinator's init function. A container with name "c1" has the following initial representation in the meta model:

[`name`:"c1", `entries`:[ ], `coordinators`:[ ], `coxts`:[ ]]

For each specified coordinator as well as for the implicitly added `SystemCoordinator`, a registration entry is added in the `coordinators` property, which includes the coordinator reference, its id and a flag indicating if it is obligatory or optional. Then, the initial container CoXTs, which are obtained by calling the coordinators' `init` functions with the specified arguments, are added at the relative path `coxts/`*`coordinatorID`* for each coordinator.

destroyContainer3: With `destroyContainer3`, a container specified by its reference is completely removed from the space. Due to the locking semantics of CAPI-2 this is not possible if other transactions still access this container. Conversely, other CAPI-3 operations cannot use this container any more after this method is called unless the operation is rollbacked.

lookupContainer3: This method finds a container by its name and returns its reference if successful. This is achieved by issuing a query on the `name` property of all container entries of the space's `containers` xtree.

setContainerLock3: An exclusive lock can be set on the `entries` property of the specified container by using the `setExclusiveLock2` operation, which prevents any read, take or write operations on this container for other transactions. A lookup on this container, however, is still valid as it does not access any entries. E.g., this behavior can be useful if an application must guarantee that no additional entries are written to a container during a transaction.

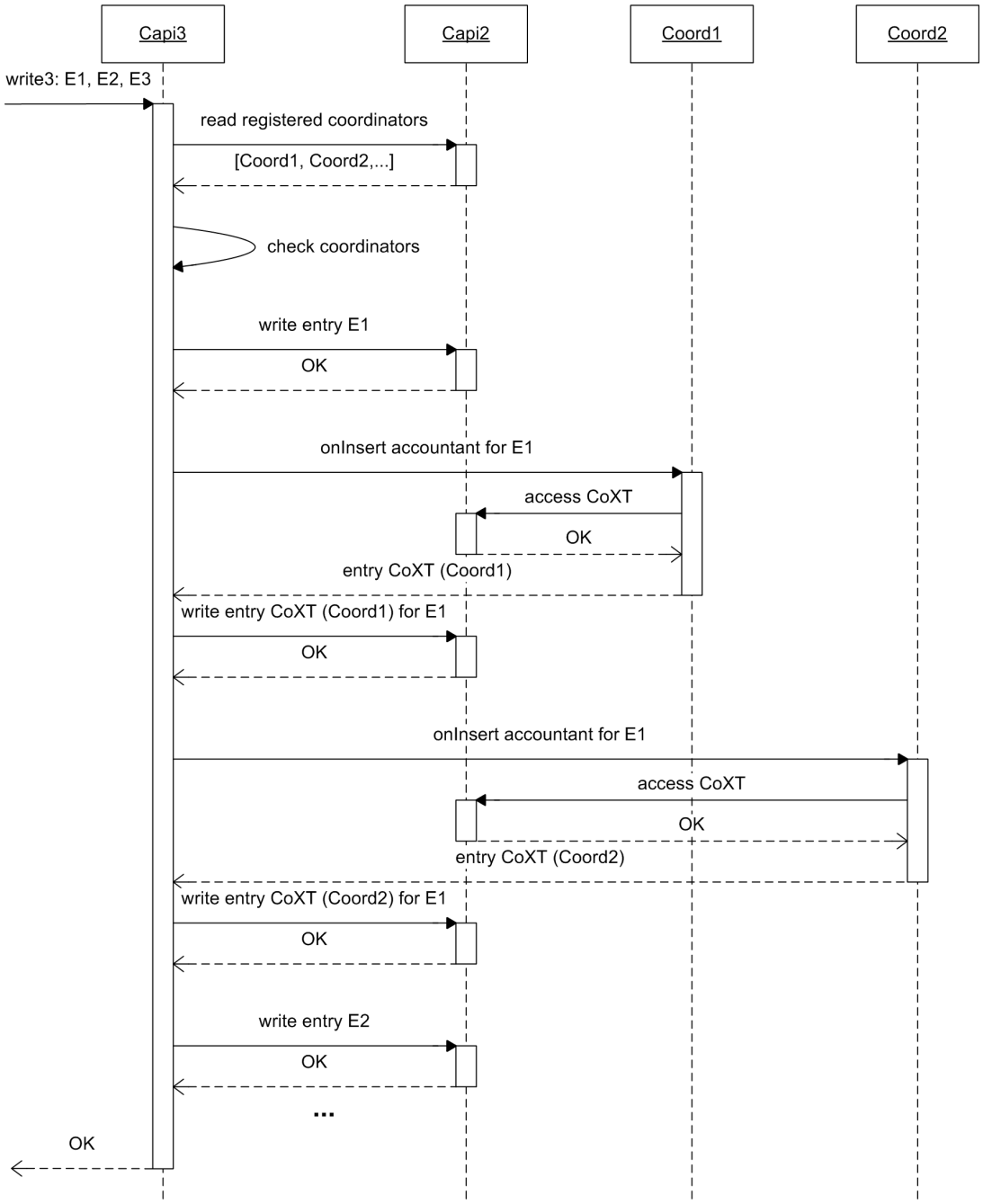

Figure 3: Writing entries with write3

write3: The arguments for `write3` contain a list of entries including write selector information that should be added to the specified container. The write selectors are given as a list of tuples with the coordinator id and the arguments to the coordinator's `onInsert` accountant function. Figure 3 shows how this function is implemented. At first, it must check that all specified coordinators are indeed registered on the container, otherwise an error is indicated. For obligatory coordinators, a write selector must only be specified if arguments are required, otherwise an empty argument xtree is implicitly used. Optional coordinators that should manage the written entries must be explicitly specified, even if their `onInsert` accountant functions do not take any arguments. If an obligatory or optional coordinator requires arguments that are not specified by the user, the accountant function indicates an error by returning status `NOTOK`. For each entry `E`, the following data structure is written into the container's `entries` property:

$$[\texttt{data}{:}E, \texttt{coordinators}{:}[\,], \texttt{coxts}{:}[\,]]$$

The actual user entry is included in the `data` property, while the names of the obligatory and all specified optional coordinators are registered in the `coordinators` xtree. For each of these coordinators, the `onInsert` accountant function is called, which may access the corresponding CoXT with several CAPI-2 read, take and write operations. The returned entry CoXT is stored in the `coxts` property using the coordinator id as label. If any accountant function returns a status other than `OK`, the execution stops and the status of that coordinator is returned to the runtime, which does a rollback of the sub transaction and possibly reschedules the corresponding request.

take3: Similar to the write selectors for `write3`, the read selectors used for this method are given as a tuple list of coordinator ids and arguments to the coordinators' query functions. The implementation of `take3` is shown in Fig. 4. If all specified coordinators are registered on the container, the query evaluation is started by calling the coordinator query function associated with the first read selector. The returned entries are then used as input for the second query function, which returns a subset of the original result and so forth. The last query function finally returns the result entries of the query. For each of these entries, the `onRead`, `onDataReturn` and `onRemove` accountant functions are called in the mentioned order for all coordinators that are registered in the entry's `coordinators` property. Thus, every coordinator that has been used to write an entry also gets informed when this entry is removed from the space. Similar to the `onInsert` method for write operations, these accountant functions may access the CoXTs of their coordinators via CAPI-2, but these calls are omitted in the figure due to limited space. The entries that are finally returned are enriched with the meta data supplied by the `onDataReturn` function, which are stored in a separate meta property for each coordinator per entry. Before

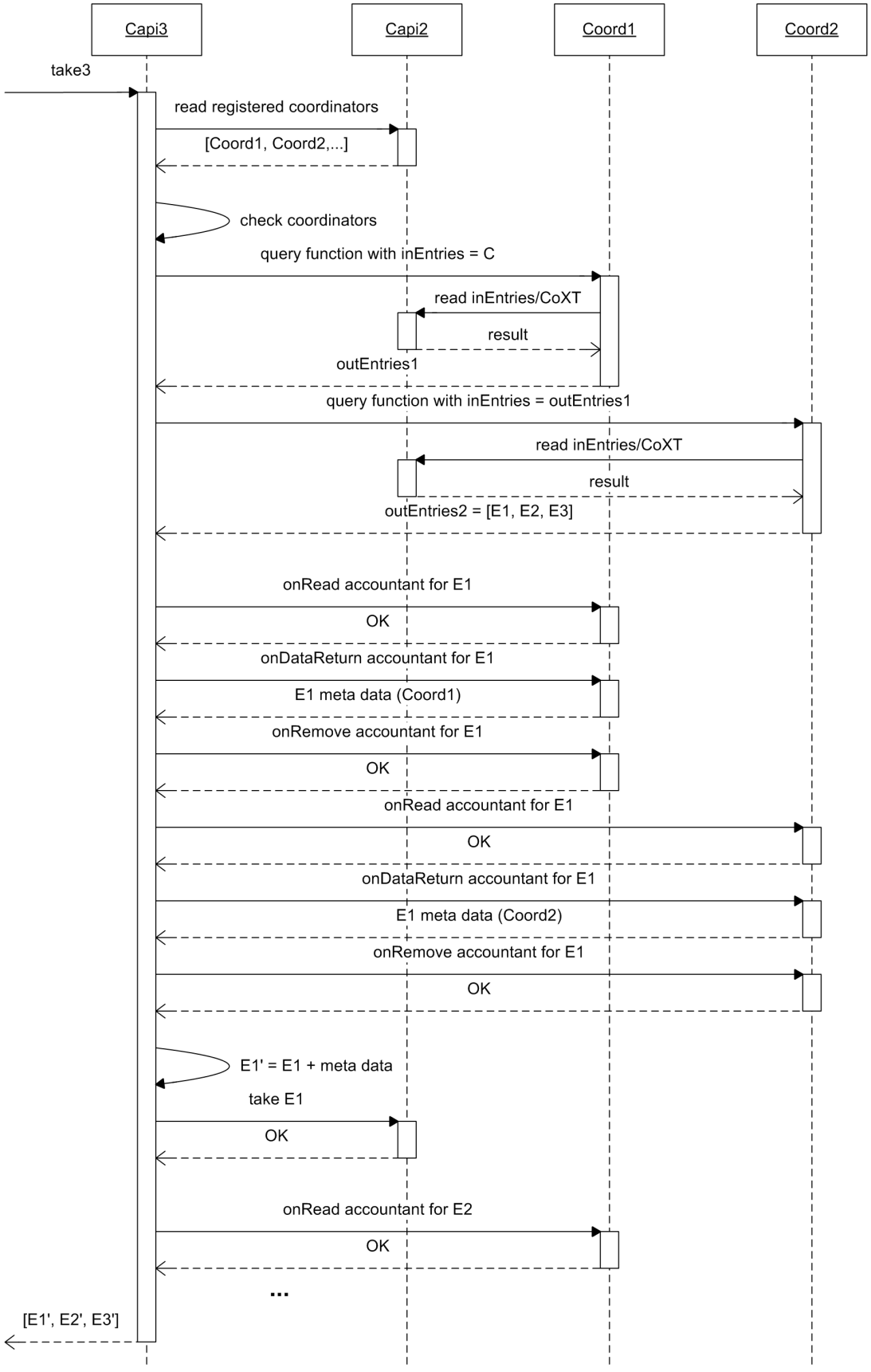

Figure 4: Taking entries with take3

`take3` returns, the result entries are finally removed from the space with `take2` because the query function has only acquired read locks on them. If this take operation fails due to locks or if any of the accountant functions indicate an error, the corresponding result xtree is returned to the runtime which handles this error.

Due to the use of CAPI-2 methods in the implementation of `take3` as well as in the coordinators' query and accountant functions, the locking behavior of this operation has to be examined closely. Each coordinator may access its own container CoXT as needed, but to allow maximal concurrency all accountant functions should only read and change as small parts of the CoXT as possible for each invocation. If a coordinator has to replace the whole CoXT every time an entry is written or taken, other transactions may not use this coordinator on the given container concurrently as any read access on the coordination data would lead to status `LOCKED`. For some coordinators, this is necessary to guarantee their integrity, e.g. concurrent access on a vector coordinator should be prohibited because otherwise duplicate or missing indices could occur. However, often it is sufficient to only lock a child xtree of the CoXT associated with a special entry, allowing the concurrent use of the coordinator at least on different entries of the container. Finally, some coordination mechanism like random or label access do not require changes in the common container CoXT and enable full concurrent access as long as the entries themselves are not yet locked by another `take3` operation. The access on the container entries within the coordinators' query functions is limited to `read2`. This means that the first coordinator puts read locks on all entries selected by its query function, and even though subsequent coordinators remove some of these entries from the result set the locks remain. The current transaction model does not allow locks to be removed before commit and the possibly complex query semantics of coordinators disallow that the whole coordinator chain is executed in a single CAPI-2 query. Therefore, often occurring combinations of coordinators could be integrated in a single composite coordinator if better concurrency is required. Another solution would be to redesign the coordinator semantics, so that they may only access their container CoXT directly with CAPI-2 methods but not the actual entries. Instead, the retrieved data from the CoXT and the arguments of the query function are used to form an XVSM query, which is then returned to the CAPI-3 method. After all coordinators have been called, CAPI-3 could concatenate the results to issue a single CAPI-2 query. These modified semantics, however, restrict the capabilities of coordinators, as any query must be independent of the actual data. E.g., it would not be possible to look for entries with a special property in their entry CoXT, then read the container CoXT using a query that depends on the first result and finally select a subset of the found entries according to the result of this query.

read3: This operation has very similar semantics to `take3` and also uses the same parameters. The only differences are that the `onRemove` accountant function is not called and that the result entries are only read but not removed from the container.

delete3: Like `take3`, this function removes selected entries from the space, but the entries are not returned. Therefore, the `onRead` and `onDataReturn` accountants are not called. Apart from that, `delete3` has the same semantics as `take3`.

4.3.3 Predefined coordinators

SystemCoordinator: This coordinator, which is implicitly added to all user containers, monitors the boundedness feature of the container and enables query access on an optionally definable amount of entries. Its CoXT is initialized with a `bounds` property that stores the specified container limit and a `size` property with value 0. However, if the container is unlimited, the size does not have to be stored. For bounded containers, the `onInsert` accountant function takes the `size` entry from the CoXT and compares it to the `bounds` entry. If the current size is equal to the container limit, the accountant function returns a result with status `DELAYABLE` indicating that the container is full. Otherwise, a new `size` entry is written with an incremented integer value. Similarly, the `onRemove` accountant function decrements this property. In both cases, the delete locks on the size entry, which are created by `take2`, prevent other transactions from writing or removing entries concurrently for bounded containers. Without this practice, however, it could not be guaranteed that the limits are never exceeded. For better concurrency, unbounded containers should be used whenever possible. A simple implementation approach that allows for more concurrency would just count the current number of entries in the container as returned by a `read2` query. However, for two uncommitted write operations in separate transactions, the respective accountant functions would not be able to see the entries written by each other. If the current container size is one less than the limit, both accountant functions would allow the operation. Thus, when both transactions commit, the container would exceed its bounds by one, which is not correct.

The query function defined by the `SystemCoordinator` requires one integer argument that specifies the number of arbitrary entries that should be returned. Apart from any positive integer, the constants `COUNT_ALL` and `COUNT_MAX` are allowed values. If a specific number `n` is given, the coordinator tries to retrieve `n` arbitrary unlocked entries using the query `cnt(n)` on a filtered version of the container that only contains entries included in the result of `getReadable2` or `getTakeable2`, respectively. The query function can determine which of these functions should be invoked by accessing the `queryType` property, which is added to the operation context by the `read3`, `take3` and `delete3` methods that call the coordinator's query function. If this query on the filtered container fails because

not enough entries are found, a second query is invoked on the entire container including locked entries, which of course also fails. However, this procedure is necessary to determine if the function result should be `LOCKED` or `DELAYABLE`. When using `COUNT_ALL` as parameter, all currently visible entries of the container are returned, which of course does not include entries that have been written in a concurrent uncommitted transaction because of the transaction semantics of CAPI-2. If any of the entries cannot be accessed due to a lock, an error is indicated with result status `LOCKED`. For a container without any entries, an empty result set is returned. Similarly, the `COUNT_MAX` argument value indicates that all available entries should be retrieved. This means that all locked entries are filtered out with the help of `getReadable2` or `getTakeable2`.

QueryCoordinator: Using this coordinator, a user can directly issue XVSMQL queries on the container. As the query function does not require any additional data apart from the container content, the accountant functions do not have to perform any actions. The query is carried out by invoking a `read2` operation on all container entries. The user query may, however, only access the actual user entries contained in the entries' `data` properties and no coordinator meta data. Therefore, the actually applied query is created by implicitly prepending `*/data` to all paths included in the query. E.g., the user query `publication/date > 2000 | cnt(1)` gets mapped to `*/data/publication/date > 2000 | cnt(1)`.

FifoCoordinator: With this coordinator, first-in-first-out (FIFO) queues as well as last-in-first-out stacks (LIFO) are supported. A precise order is achieved by attaching a consecutive number to each written entry that indicates the sorting of the container. To prevent concurrent transactions from writing an entry with the same index to the container, the CoXT consists of a `nextIndex` property which initially has a value of 1 and is incremented for each written entry by the `onInsert` accountant function. The obtained value is then used to attach a `writeIndex` property to the entry CoXT. Concurrent write operations are not possible with this implementation because of the delete lock on the `nextIndex` property needed for this update. The removal of entries, however, does not block other transactions because the relative order stays the same even if the sequence of indices contains gaps. A completely concurrent alternative could be implemented by using a timestamp parameter which is passed to the coordinator by the runtime via the context. The timestamps, which could also be stored in the entry CoXTs, indicate an order too. However, this order is not unambiguous as two entries A and B could be written at the same time. In this case, entry A might be returned before entry B in FIFO mode as well as in LIFO mode, which is not feasible because LIFO should use the reverse order of FIFO.

The query function of this coordinator takes two arguments. A mode parameter specifies if FIFO or LIFO order should be applied, while a count parameter specifies the number of elements that should be returned. In FIFO mode, the `sortup` predicate is used in a query on the `writeIndex` property of each entry, while in LIFO mode `sortdown` is applied. The count parameter specifies how many entries should be returned in the specified order using the `cnt` selector. If `COUNT_ALL` is specified, all managed entries are returned. Unlike the `SystemCoordinator`, locked entries are never filtered from the input entries because otherwise a strict FIFO or LIFO order cannot be guaranteed. This means that two concurrent take operations using the same mode are not possible because both transactions try to lock the same entries exclusively. However, one FIFO and one LIFO take operation might be able to be executed in parallel if the result set is not overlapping. A variant of this coordinator could be built that only selects unlocked entries, which could mean that in FIFO mode an entry B is selected even though another (locked) entry A written before B exists in the container. However, if the transaction that has locked entry A performs a rollback, the first transaction might read entry A after entry B and thus in the wrong order.

KeyCoordinator: This coordinator uses a unique key to retrieve an entry. The CoXT stores xtrees for each entry containing the key and the label of the entry within the container, which is used as a reference. The `onInsert` accountant requires a string argument specifying the key for the entry. It checks whether this key already exists in the CoXT and if such an entry is found, a `DELAYABLE` result is returned. Concurrent write access on this CoXT must be prevented because otherwise duplicate keys could be established. However, concurrent access via read, take or delete should be possible while new entries are written to the container. Therefore, a special `writeLock` property exists in the CoXT which must be locked by the `onInsert` function with `setExclusiveLock2`. The `onRemove` accountant takes the xtree associated with the deleted entry from the container CoXT via a matchmaker comparison of the entry's label with the references in the CoXT. Similarly, the `onDataReturn` accountant reads the key of an entry from the coordinator's data structure and adds it to the meta data that are returned to the user.

The query function, which requires a string parameter representing the key, retrieves an entry in two steps. At first, the CoXT is queried for an xtree with the specified key value. If none is found, the function returns `DELAYABLE`. Otherwise, the entry reference given in the retrieved xtree is used for a second query on the actual container and the entry with the corresponding label is returned.

LabelCoordinator: Similar to the `KeyCoordinator`, entries are accessible via a special label which, however, does not have to be unique. Therefore, multiple entries can be

stored using the same label. As the check for duplicates is omitted, the coordinator does not require a central data structure to synchronize write access. Instead, the `onInsert` accountant attaches the label directly to the entry CoXTs in a property named `lbl`, which allows full concurrent access. Therefore, removing an entry does not require any action by the coordinator, whereas the `onDataReturn` accountant adds the label to the entry's meta data.

For queries, the label and a count parameter must be specified. Then, as many entries with this label as specified are selected, which is achieved via a matchmaker comparison on the `lbl` property of the corresponding entry CoXTs. The semantics of the count parameter are the same as for the `SystemCoordinator`, thus the filtering of locked entries is possible.

LindaCoordinator: The Linda template matching mechanism is supported by this coordinator. The accountant functions do not require any additional meta data apart from the actual data fields. Nevertheless, the `onInsert` accountant function has to check if the written entry has the correct format for this coordinator, which basically is a sequence xtree with arbitrary content. The query function matches the specified template sequence with the managed entries and returns as many fitting entries as specified by the count parameter, which has the same meaning as in the `SystemCoordinator`. Thus, in contrast to the classical Linda model, more than one entry can be read at once. An entry matches the template if both have the same number of properties and if every property not marked with a wildcard in the template has an equal value in both sequences. For this coordinator, only the position of the properties within the sequence is relevant and the property label is ignored, although different versions could be built that also compare the labels. For comparison, the following recursive matchmaker function is used on the value of the entry `data` property (`E`) with template `tmpl`:

$$
\begin{aligned}
&lindaMatchmaker(tmpl, E) := \\
&\quad |tmpl| = |E| \ \wedge \ (|tmpl| = 0 \ \vee \ ((tmpl.1 = \text{“*”} \ \vee \ tmpl.1 = E.1) \\
&\qquad\qquad \wedge \ lindaMatchmaker(rest(tmpl), rest(E))))
\end{aligned}
$$

VectorCoordinator: This coordinator presents its managed entries as vector that can be accessed via index positions starting with 0. As each written or removed entry may shift the positions of other entries, any operation other than read must lock the whole coordinator. This is done with `take2` and `write2` operations on the `positions` property in the CoXT. This data structure models the vector by storing the entry labels used to reference the actual entries in a sequence xtree. The entry corresponding to the first reference in this sequence has vector position 0, the second reference corresponds to index

1 and so forth. The `onInsert` function requires an argument specifying the position of the new entry within the vector. The constant value `APPEND` as position argument indicates that the entry should be added at the end of the sequence. If an invalid index is specified as parameter, the result status `NOTOK` is returned. The entry reference is inserted at the corresponding index of the `positions` sequence, which is then written back to the CoXT. Similarly, the `onRemove` accountant function updates this xtree by deleting the entry label of the removed entry. The `onDataReturn` function retrieves the current vector index of the entry by searching for the label in the `positions` sequence.

The query function uses the XVSMQL query `cnt(i+1) | reverse() | cnt(1)` at the `positions` xtree of the CoXT to get the reference at the index `i`, which is specified as a parameter. Then, the found label is used in a query on the actual container to get the searched entry. As for write, an invalid index leads to a `NOTOK` error.

4.3.4 Custom Coordination

Apart from the predefined coordinators, user applications may use custom coordinators that are specifically designed for a particular use case [24]. These user coordinators can be invoked just like the predefined ones, as long as their implementation is registered on the runtime in the `coordinatorDefs` xtree. This possibility enables the developer to strictly separate the application logic from coordination mechanisms and therefore encourages a clean design [33]. By using CAPI-2, the coordinator does not have to handle transactional integrity by itself, as all actions are automatically logged in a transaction. To allow optimal concurrency, however, a coordinator developer should be familiar with the locking semantics of CAPI-2, so that parallel transactions using the same coordinator are only blocked if required by the coordination semantics.

One example for this flexible coordination mechanism is a priority queue for entries corresponding to tasks, as described in [24]. There, the entries are managed by a special `PrioFifoCoordinator` which sorts the tasks by priority, whereas for equal priority a FIFO order is used. It has been shown that such a coordinator greatly simplifies the data access compared to the classical Linda template matching approach.

4.4 CAPI-4: Runtime model

The XVSM runtime, also known as CAPI-4, represents a single XVSM core that can be accessed by any kind of language API, either directly in embedded mode or over a network using the XVSMP layer. Each core corresponds to one XVSM space which is used to bootstrap the runtime. Within the meta model of the runtime, operations that should be invoked on the space are embodied as *request xtrees* consisting of the operation name and arguments as well as meta information added by the runtime and the URI of

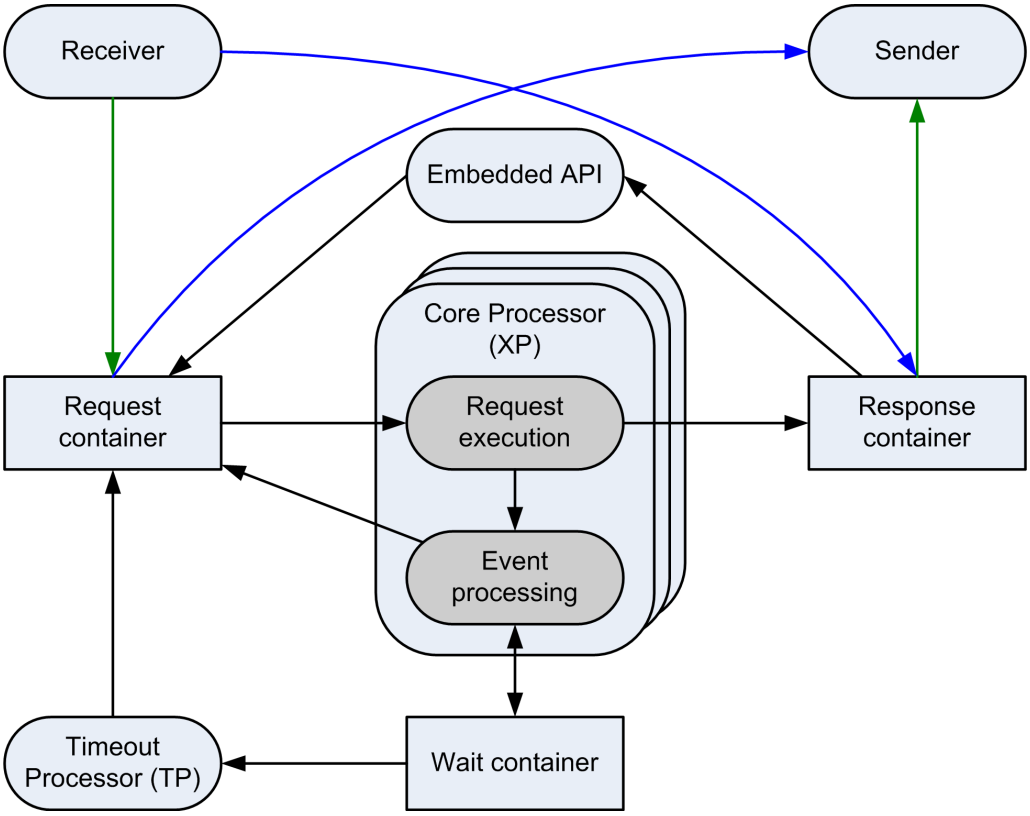

Figure 5: XVSM runtime structure

the caller. Figure 5 shows the basic architecture of a single XVSM core, which consists of several runtime threads and meta containers. Each request is initially put into the *request container*, either directly via the *embedded API* or after the *receiver process* has received a remote XVSMP message, which is then transformed into a request xtree. For each new request, a unique id is generated and returned to the invoking process, which is used to retrieve the result. The most important part of the runtime are the *XVSM core processors* (XP), which are concurrently executing active requests and writing their results into the *response container*. From there, the embedded API may get the result either by waiting synchronously on an xtree with fitting id or by polling for the result asynchronously. Similarly, the sender process may wait for any result dedicated to a remote caller, which is then marshalled into XVSMP format and sent to the specified address. Thus, there are three possibilities how a request passes through the runtime:

- For local requests on the own space, the embedded API is used directly, which writes a request into the request container and retrieves the result from the response container.
- When the embedded API invokes a remote space, the sender process takes the request from the request container, whereas the receiver writes the remote XVSM core's answer to the response container (blue arrows). From there, the embedded

API can take the result.

- When a remote API calls the local runtime, the receiver puts the request into the request container, while the sender waits for any response dedicated to a remote peer and then sends it to the specified URI (green arrows).

The core processor executes the request by calling the appropriate CAPI-3 function and the associated aspects and analyzing the result. Aspects can be executed either before the call to the actual CAPI-3 operation (*pre aspects*) or afterwards (*post aspects*). They are able to modify parameters, including the context xtree, and return values and may invoke own actions on the space or on other available resources. If the core processor obtains a result with status `OK` or `NOTOK`, the result xtree is immediately written to the response container. After the actual execution of the request, the core processor invokes the *event processing logic*. If status `LOCKED` or `DELAYABLE` is returned, the runtime may reschedule the request by putting it into the *wait container*. For any of these result status codes, the event processing logic checks the wait container if the events generated by the current request invocation are able to wake up sleeping requests. In contrast to earlier versions of the XVSM runtime [43, 44], the events need not be stored in an own container because the wait and event containers are combined in a single wait container (see Sect. 4.4.1).

If a request is rescheduled, it is put back into the request container so that the core processors are able to process it again. The *timeout processor* (TP) periodically checks the wait container and removes requests with expired timeouts. These requests are also rescheduled and reprocessed by a core processor, but the timeout is immediately recognized so that an error message is generated before any actual call to CAPI-3 or any of the aspects. The status code for this request result is `INVALID`, which has the same meaning as `NOTOK`, except that requests marked as `INVALID` cannot trigger any events because they do not invoke any CAPI-3 method. Thus, the result xtree for a timed out request is also written to the response container by the core processor.

The meta data structure of the runtime is accessed directly via a blocking variant of CAPI-1 called CAPI-B. Basically, this CAPI uses the same operations as the original data access layer but offers the additional blocking versions of `read1` and `take1`, which are named `readB` and `takeB` and use the same parameters. The semantics of these versions are to block until the operation yields a return status of either `OK` or `NOTOK`. If the status would be `DELAYABLE`, the operation is simply retried when something changes within the accessed data structure. This allows the core processor to wait for new tasks in the request container by using a blocking take query on it instead of constantly polling the container with `take1`.

4.4.1 Request scheduling

The processing of a request starts when the `writeRequest` method of the runtime is called which writes the specified request xtree into the request container at location `_reqC` within the meta model and returns a unique request id that is also added to the request xtree. All idle core processors use a `takeB` query on this container to fetch a new request for execution. The XP that gets the request then adds a `timestamp` property to the xtree using the current system time if it is entering the core for the first time, i.e. it was not yet rescheduled by the runtime. Depending on the value of the request's `timeout` parameter, also the `expireTime` property may be set which determines when the request should expire. Following values are valid timeouts, with `TRY-ONCE` used as default if no timeout is specified:

- **Integer > 0:** Any positive value can be defined, which specifies the minimum amount of milliseconds the request remains valid after it enters the core for the first time. The expire time is set to `timestamp + timeout` and if this point of time is exceeded when the entry is fetched, the request fails and an `INVALID` response xtree is generated which indicates the timeout. There is, however, no guarantee that the timeout occurs exactly after the specified time as the timeout processor only checks periodically for expired entries and the core processors may be busy which might delay the response.
- **INFINITE:** This constant indicates that the request should never expire, so it should always be rescheduled if the result is `DELAYABLE` or `LOCKED`.
- **TRY-ONCE:** This value specifies that the operation should have the opportunity to acquire all necessary locks and check if this request is currently satisfiable or not. As long as the operation yields a `LOCKED` result, it should be rescheduled. If, however, status `DELAYABLE` is returned, an error response is generated.
- **ZERO:** This constant indicates that the request must never be rescheduled and is only tried to process once. Any result status other than `OK` leads to a `NOTOK` response.

Additionally, a `lastExecutionTime` timestamp is updated every time the request enters the core, which is later needed by the event processing logic that handles the rescheduling of waiting requests. If the request has not yet expired, the actual request execution can start. It must be noted that for this check as well as for the mentioned timestamps the time value obtained when fetching the request from the request container is used. Therefore, a request that exceeds its timeout during execution is still valid.

The core processor distinguishes between operations called within a transaction and the control operations `createTransaction`, `commitTransaction`, `rollbackTransaction`, `addAspect` and `removeAspect`. These control operations do not support a transactional context, while the remaining tasks require a transaction and operation id. The transaction id is usually given by the user but it can be omitted, which indicates an implicit transaction for this single request. In this case, a new transaction is created by the runtime with `createTransaction2` and it is committed after the request execution has completed. The operation id is always supplied by the runtime, which opens a new sub transaction for the given user transaction by calling `startOperation2`. Depending on the result of the request execution, the operation is completed. If status `OK` is returned, the runtime invokes `finishOperation2` and thus commits the sub transaction. For any other result status, `cancelOperation2` is called which rollbacks all changes made to the space during the execution of the request.

The request is executed by calling all pre aspects of the operation, then the corresponding CAPI-3 operation itself and finally the associated post aspects. The result xtree of this invocation includes a `subOps` property that lists all used CAPI-3 operations as the aspects may have called arbitrary CAPI-3 operations different from the one listed in the request. If the result status is `LOCKED` or `DELAYABLE`, an additional property `waitForOp` is needed that indicates the CAPI-3 operation that has failed and thus led to the abortion of the operation. Both properties are later used for the event processing logic that needs to know which operations were executed and why a request must wait. If the result status is `OK` or `NOTOK`, the response xtree, which consists of the request id, the URI of the caller and the operation result, is written to the response container at location `_respC`, from where it can be retrieved either synchronously with `takeB` or asynchronously via the non-blocking `take1`.

In the following example, a request xtree for a take operation on container `C` using a FifoCoordinator read selector is shown, as well as its corresponding response xtree. The missing properties for the URIs of the caller and the space indicate that the local embedded API has issued this request on the local space, which means that receiver and sender processes are not involved.

`Request` : [`id`:76, `op`:"take", `cref`:C, `selectors`:$\langle fifo(1)\rangle$, `txid`:4,
`timeout`:10, `timestamp`:126, `lastExecutionTime`:131, `expireTime`:136]

`Response` : [`id`:76,
`result`:[`status`:"OK", `entries`:$\langle E1\rangle$, `subOps`:$\langle$[`op`:"take", `cref`:C, `txid`:4]$\rangle$]]

After the actual result execution, the event processing logic is invoked by the XP unless the request result is `INVALID`. The basic idea of this routine is that any time a request

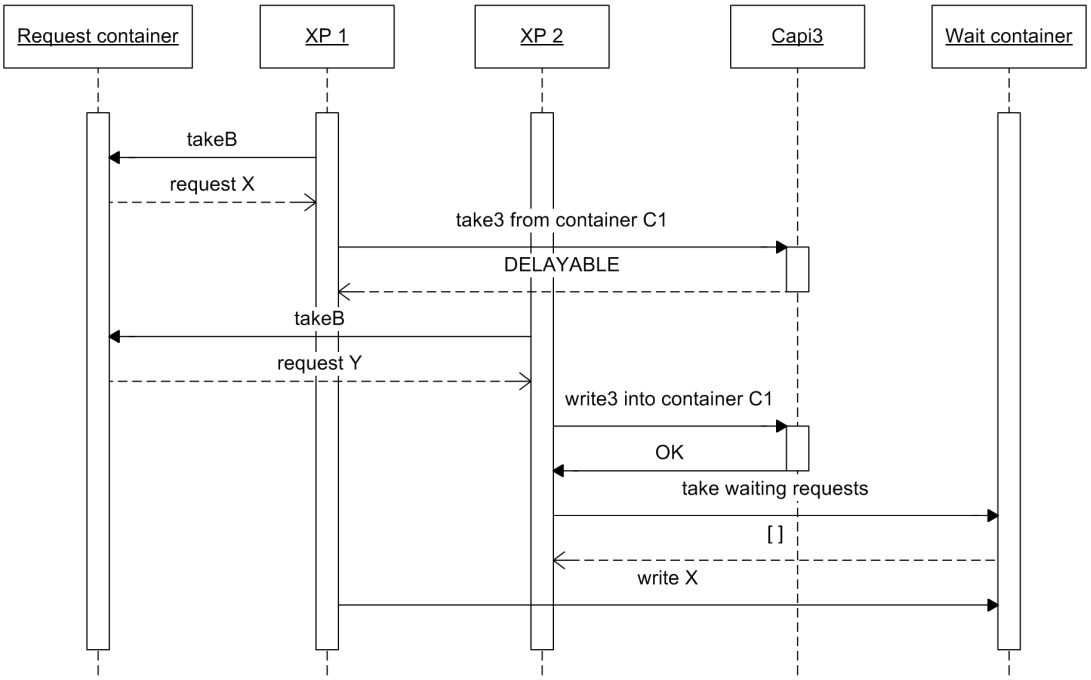

Figure 6: Race conditions when using simple event processing

has been executed, all waiting requests that can be possibly woken up by the performed changes are rescheduled. So, if the result is `DELAYABLE`, a read, take or delete operation must wait until new entries are written to the corresponding container, whereas a write operation should be rescheduled when something is taken or deleted from the container because entries that are in conflict with the write operation could have been removed. For results with status `LOCKED`, the request has to wait until some other request releases the locks, which usually happens at transaction commit or rollback. Waiting requests are stored in the wait container (`_waitC`) and every operation result is compared with these sleeping requests to check if something has to be rescheduled. In this case, the requests are taken from the wait container and written to the request container. There are, however, some problems with this simple approach that need to be solved:

- First of all, events can be lost if two or more requests are processed concurrently by different core processors. This problem is illustrated in Fig. 6, where a take and a write request on the same container are executed in parallel by the core processors `XP1` and `XP2`. A request `X` may access the space and obtain a `DELAYABLE` or `LOCKED` result before another request `Y` invokes changes on the space but the event processing logic of `Y` might be called before `X` is written to the wait container. Thus, the event generated by `Y` is not aware of `X`, even though the changes might enable its successful execution. When `X` is finally put into the wait container, the event related to `Y` has already been processed and therefore the request sleeps although it should be rescheduled. To solve this problem, not only the waiting requests must be stored

but also the events generated by completed requests. However, to prevent that the list grows infinitely, events must be removed from the space after they cannot wake up requests any more. This garbage collection implies additional complexity and therefore a different approach is used. Instead of storing the events individually, categories are created that store timestamps for when an event of a specific type on a particular container has occurred the last time. Each waiting request is put into a queue in exactly one of these categories, while any completed request execution leads to an update of one or more event timestamps.

- It is further necessary to distinguish between committed events and those that are still part of an open user transaction. In the latter case, only waiting operations of the same transaction should be woken up as the others cannot yet see the changes made by the operation that has caused the event.

- Usually, only successful requests with status `OK` lead to events that are able to wake up waiting requests. Any operation accessing the space, however, may set volatile locks within its sub transaction before the operation is cancelled and the locks are released. These locks may hinder concurrently executing operations from fulfilling their task and therefore also an event must be generated for unsuccessful requests to wake up those requests with status `LOCKED`. If a successful request `Z`, which is part of an active transaction, holds a lock that is also needed by several other requests, these requests are all put into the wait container waiting for an unlock event. If the requests are later rescheduled and executed concurrently, they may also acquire locks before they fail due to the lock held by `Z`. Therefore, they invoke an unlock event after the operation is rollbacked, while waiting themselves for an unlock event. To prevent that the requests wake up each other every time their execution fails, causing a possibly endless chain of mutual rescheduling, the event processing logic should distinguish between *short-term locks* that are part of an active sub transaction and *long-term locks*, which are valid until the user transaction commits or rollbacks. The runtime can differentiate these lock types by looking at the `lockType` property that CAPI-2 adds to its result xtree, which indicates if a result status `LOCKED` was caused by a short-term lock of a still active sub transaction or by a long-term lock that is directly held by the user transaction. In the mentioned example, the blocked requests would wait for an unlock event on long-term locks, while they generate an unlock event for short-term locks themselves. Therefore, they are not able to wake up each other and busy waiting is avoided. However, it could occur that two requests are blocked on different short-term locks when one of these acquires a lock that the other one needs and vice versa. If the requests are always started simultaneously, they both return with status `LOCKED` and generate an unlock

event for short-term locks so that they wake up each other again. Therefore, the requests waiting in this category are not executed in parallel but only rescheduled as a sequential list.

For each new user container with reference `cref`, the following data structure is generated in the meta model:

cref : [`insert`:[`lastCommittedTime`:0, `uncommittedTime`:[ ], `waitingRequests`:⟨ ⟩],
`remove`:[...],
`lt_unlock`:[...],
`st_unlock`:[...]]

For status `DELAYABLE`, write operations are associated with the `remove` category, while read, take and delete operations use the `insert` category. The `lt_unlock` and `st_unlock` categories are used for tasks that are locked on a long-term or short-term lock, respectively. As long-term locks always evolve from short-term locks, any event triggering a `lt_unlock` event must also trigger `st_unlock`. Otherwise, a request that tries to acquire a lock before the corresponding sub transaction has committed and therefore waits on a `st_unlock` event would not be notified about the freed data structure. Whenever a request execution is finished with result status `LOCKED` or `DELAYABLE`, the event processing logic performs the following steps:

1. Read the `waitForOp` property which contains the operation type, the container and the transaction of the CAPI-3 operation that has led to this result.

2. For the thereby obtained category, read the `lastCommittedTime` timestamp and, for transactions that are not implicit, also the timestamp of the operation's transaction in the `uncommittedTime` multiset.

3. If any of these timestamps is greater than the `lastExecutionTime` of the request, the request is immediately rescheduled because an event exists that was fired after the execution of the request has been started. Thus, the operation might have been executed before the changes associated with the event were made to the space. Otherwise, the request is put into the `waitingRequests` xtree of the corresponding category.

Then, for any result status, the appropriate events are triggered for all operations that are part of the request, as specified in the `subOps` property:

1. For all categories that are affected by the operation, update the timestamp entry for the specified transaction in the `uncommittedTime` multiset with the current

time, using the transaction id as property label. If the operation status is not `OK` and thus a generally visible `st_unlock` event must be triggered or if the operation uses an implicit transaction that automatically commits, the `lastCommittedTime` timestamp is changed instead. When the request contains a `commitTransaction2` operation, all timestamps of this transaction in the `uncommittedTime` xtrees of all categories are removed and the corresponding `lastCommittedTime` properties are updated with the current time. For `rollbackTransaction2`, a similar strategy is used, but only for the categories of type `st_unlock` and `lt_unlock` because the effects of `insert` and `remove` events are revoked. Additionally, all currently waiting requests of a completed transaction must be rescheduled, as their transaction is now invalid.

2. For all changed timestamps, check the `lastExecutionTime` of waiting requests in the corresponding categories. If the new event timestamp is greater than the timestamp of the request and if the event is visible for the request's transaction, the request is rescheduled.

Figure 7 shows an example for the functionality of the event processing logic. A take request `X` on container `C1` is not able to succeed and is therefore put into the `C1/insert` category of the wait container. The attempted execution triggers an `st_unlock` event, which does not wake up any request. The subsequent request `Y`, which depicts a write operation on the same container with an implicit transaction, is executed successfully and `X` is rescheduled because the event timestamp for the `insert` category is updated.

Due to the non-blocking semantics of CAPI-2 and 3, this runtime machine may never run into a deadlock. It is, however, possible to get stuck in a livelock when two transactions require locks that have been mutually acquired by previous operations of these transactions. The current solution to this problem are transaction timeouts. The user can specify how long a transaction is valid. Upon creation of the transaction, the runtime writes the expiration time into a special runtime container called `_txTimeoutC`. The timeout processor checks periodically if any transaction is expired. In this case, the transaction is rollbacked and all sleeping requests of this transaction are invalidated. The second task of the timeout processor is to reschedule requests with expired request timeout. This is also done periodically by issuing a query on the entries of the `waitingRequests` xtree of all categories.

The presented approach for the semantics of the event processing logic is only one of many possible solutions. On the one hand, requests should not be woken up if they cannot be fulfilled, but on the other hand, the decision whether a request should be rescheduled should not require extensive computations. Therefore, a balance must be found between accuracy and fast decisions to achieve optimal performance. Any feasible approach must

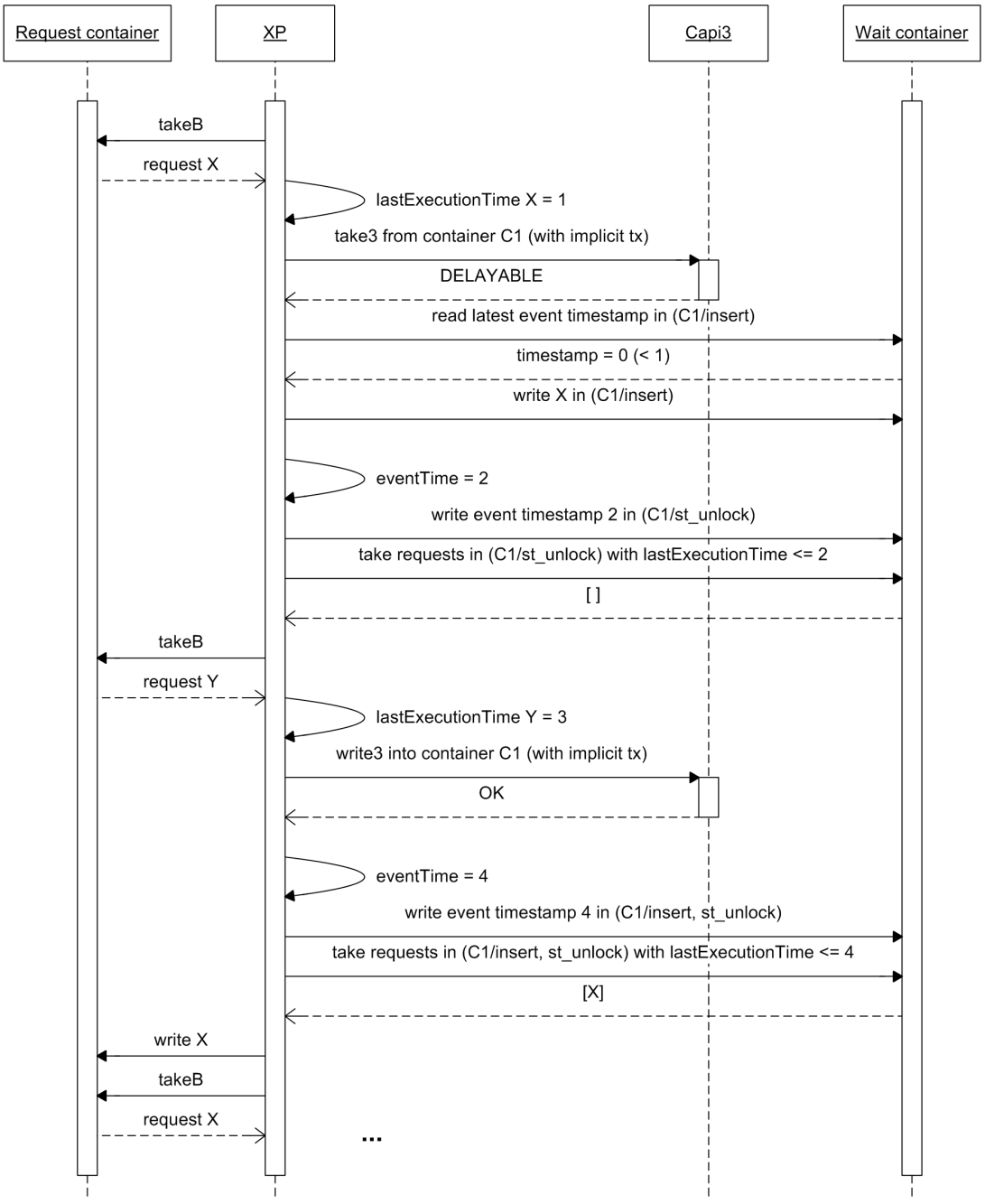

Figure 7: Event processing logic

reschedule requests immediately when they are able to execute, but requests could also be woken up if there is only a chance that they can be fulfilled. Requests that are unnecessarily woken up do only affect performance and not the semantics of the runtime, as they are just put back into the wait container when their execution fails again. A very simple event processing logic would just reschedule all requests of a container after an operation on this container has been processed. The specified semantics offer a more appropriate behavior by introducing categories for requests that wait for insert, remove or unlock events, without requiring complex computations or extensive storage. Another possible solution would be to split the events into even more categories, so that the event logic can determine more exact preconditions for a request to be satisfiable. E.g., for a take request with status `DELAYABLE`, the corresponding coordinator could be specified, so that the request is only woken up if new entries are written to the container using this coordinator. Furthermore, a locked request could include data on the transaction that holds the conflicting lock, so that the request is only rescheduled if the related (sub) transaction commits or rollbacks. It must, however, be examined if the added accuracy outweighs the higher computational overhead, which will be researched in future work.

4.4.2 Aspects

Aspects can dynamically change the semantics of operations. Pre aspects may change the request xtree and thus the arguments of an operation, while post aspects can manipulate the result xtree. They can also access the space directly via own calls to CAPI-3 and may have additional side effects like access to IO or databases. Aspects can be used for tasks like notifications, logging, security [42] or replication [26]. It is possible for aspects to use the same sub transaction as the actual operation so that the changes are automatically rollbacked when the operation fails. This is enabled by the runtime, which adds the id of the active operation to the context xtree that is passed to the aspects. It must be noted, however, that changes outside of the XVSM core are not transactionally safe, so aspects should invoke proper compensation if an operation fails or a transaction rollbacks. Also, the transaction and aspect control commands do not support a transactional context for their aspects because they do not use own sub transactions.

Aspects are added to the runtime via so-called *interception points*. These interception points are either before or after any CAPI-4 operation. So-called *local aspects* can be defined for write, read, take and delete operations, which are only valid on a certain container. In contrast, *global aspects* are always invoked when the associated CAPI-4 operation is called. An aspect is registered on an interception point by adding an entry with its code to the appropriate interception point sequence of the `aspectDefs` meta container. The aspect is specified via a simple scripting language, which allows easy access to the request and result parameters as well as to the space. This scripting

language, however, is out of scope of this thesis and will be specified in future work.

For every runtime operation, the pre and post aspects are called in the defined order. Their return values can have the same status codes as the CAPI-3 operations but pre aspects may have an additional status `SKIP`. This code means that all following pre aspects as well as the actual operation should be skipped and the core processor should resume with the post aspects. If no post aspect exists, status `OK` is returned for the whole operation. If any pre or post aspect or the actual operation return a status of `NOTOK`, `LOCKED` or `DELAYABLE`, the execution is immediately stopped and this value is returned as the operation result. Otherwise, the operation result of the last post aspect is used unless there are no registered post aspects, in which case the result of the actual operation is returned.

Beside the request xtree for pre aspects, and the request and result xtrees for post aspects, the context parameter is used as an additional argument that can also be modified by all aspects. This aspect context is originally provided by the user, who may pass any kind of information to the aspects. Before invoking the aspects, the runtime may also add meta data to this context. This aspect context can then be used to pass information between aspects and between a pre aspect and a coordinator which also has access to this parameter.

The aspect mechanism allows any user to enrich the space with arbitrary behavior. *Profiles* are used to add a particular functionality to the space, which may include security, replication or notifications. These profiles are implemented via a combination of aspects and are an example for the extensibility of XVSM.

4.4.3 CAPI-4 operations

In contrast to the lower CAPI layers, the methods of CAPI-4 shown in Table 7 are not called directly but invoked via the runtime machine by putting a corresponding request into the request container. The result which is finally put into the response container has the status `OK`, `NOTOK` or `INVALID`. All operations have a `SpaceURI` parameter that determines the XVSM core on which the request should be performed. If this value is null, the local runtime is used. Otherwise, the sender process takes the request from the request container and sends it to the appropriate core. The context parameter is used to pass special information to aspects and coordinators. Most of the operations also use a transaction id as an argument. A null value in this case indicates an implicit transaction created by the runtime that should be automatically committed.

Transaction management: For creating, committing or rollbacking a transaction, the corresponding CAPI-2 methods are called. The transaction timeout specified for `createTransaction` may have any value greater than 0 for a period in milliseconds or

	Arguments	**Result**
createTransaction	SpaceURI (String), Context (Xtree), Transaction timeout (Integer)	Transaction Id (String)
commitTransaction	SpaceURI (String), Context (Xtree), Transaction Id (String)	-
rollbackTransaction	SpaceURI (String), Context (Xtree), Transaction Id (String)	-
addAspect	SpaceURI (String), IPoint (Interception Point), Code (Aspect), Context (Xtree)	Aspect Id (String)
removeAspect	SpaceURI (String), Aspect Id (String), Context (Xtree)	-
createContainer	SpaceURI (String), Container name (String), Size (Integer), Obligatory Coords ([(Coord, String, Xtree)]), Optional Coords ([(Coord, String, Xtree)]), Context (Xtree), Transaction Id (String)	ContainerRef (String)
destroyContainer	SpaceURI (String), ContainerRef (String), Context (Xtree), Transaction Id (String)	-
lookupContainer	SpaceURI (String), Container name (String), Context (Xtree), Transaction Id (String)	ContainerRef (String)
setContainerLock	SpaceURI (String), ContainerRef (String), Context (Xtree), Transaction Id (String)	-
write	SpaceURI (String), ContainerRef (String), Entries with write selectors ([(XTree, [(String, Xtree)])]), Context (Xtree), Transaction Id (String), Timeout (Integer)	-
take	SpaceURI (String), ContainerRef (String), Read selectors ([(String, Xtree)]), Context (Xtree), Transaction Id (String), Timeout (Integer)	Result entries (Xtree)
read	SpaceURI (String), ContainerRef (String), Read selectors ([(String, Xtree)]), Context (Xtree), Transaction Id (String), Timeout (Integer)	Result entries (Xtree)
delete	SpaceURI (String), ContainerRef (String), Read selectors ([(String, Xtree)]), Context (Xtree), Transaction Id (String), Timeout (Integer)	-

Table 7: CAPI-4 operations

the constant value `INFINITE` for a transaction that should never time out.

Aspect management: As described in Sect. 4.4.2, aspects can be dynamically added to an interception point with `addAspect`. The returned unique aspect id can then be used to delete the aspect with `removeAspect`.

Container operations: The remaining operations on containers and their entries are executed by calling the corresponding CAPI-3 methods. The timeout parameter, which is only available for `read`, `take`, `delete` and `write`, specifies how long the execution should be tried, as described in Sect. 4.4.1.

Management API: Beside the actual CAPI-4 operations, there is also the possibility for a privileged user to directly access the XVSM meta model. This is done by allowing special requests for debug operations that essentially comply with the basic methods of CAPI-1.

4.5 XVSMP and Language Bindings

The XML based protocol that enables the interaction between different XVSM peers is called XVSMP. Within the runtime, the sender and receiver processes form the protocol layer that transforms XVSMP into the internal xtree representation and vice versa. The intermediate protocol allows XVSM implementations in different programming languages to communicate with each other, thus enabling platform interoperability. There are two types of language bindings: An embedded API as described in Sect. 4.4 communicates directly with the XVSM core via the request and response container. If a request is issued for a remote space, the runtime uses the protocol layer to obtain the result, but this remains transparent to the API. A standalone API without own XVSM runtime can also be implemented. This language binding uses XVSMP directly by sending appropriate XML messages to the URI of the space. In this case, an *answer container* must be specified where the request result and its status information is written to. Usually, this answer container remains virtual, which means that the protocol layer sends the response directly to the URI of the calling language adapter. However, it is possible to specify a particular user container located at an XVSM core. This could be useful for mobile devices that do not have a permanent connection to the space. Language adapters on these devices can issue a request that may require some time to execute and then disconnect immediately. Later, they can poll the specified answer container for the actual result. XVSM language adapters may provide synchronous access to space methods or asynchronous invocation of XVSM operations via callback functions. The protocol supports both approaches, thus allowing for maximal flexibility.

5 Application scenario

As a use case scenario for the application of the XVSM middleware, a simple auction system is presented in this section, similar to the example described in [10]. Multiple peers should be able to buy and sell books over the internet in an ad-hoc way. The participants do not have to know each other in advance. Instead, the entire collaboration between peers is accomplished via a common XVSM space. It should be possible to dynamically create auctions, search for them and make bids. Moreover, the implementation and adaptation of the individual applications should be possible with minimal effort, so that different policies for peers can be established easily. In the following, a possible design for this use case is shown; a concrete implementation of this scenario using the XVSM Haskell prototype is described in Sect. 6.3.

A single container holds all entries used for the coordination of the processes. For each auction, four different kinds of entry types are needed. An *auction entry* with a specified starting bid is created for every book that needs to be sold, while *bid entries* store one bid of a potential buyer for a particular active auction. Shortly before an auction is closed, the seller may specify a *countdown entry* to notify potential bidders. When an auction ends, all related entries are cleared and an *acceptance entry*, which specifies the winner of the auction, is inserted into the container. Auctions, sellers and bidders are all identified via a unique id. The purpose of an entry is determined by a separate field specifying the type. This property contains the strings “auction”, “bid”, “countdown” or “accept” for the respective entry types. Example entries for this scenario look as follows:

[`type`:“auction”, `auctionId`:“a42”, `startingBid`:10, `sellerId`:“seller1”,
`book`:[`title`:“JavaSpaces Example by Example”, `author`:“Halter”]]
[`type`:“bid”, `auctionId`:“a42”, `bid`:12, `bidderId`:“bidder1”]
[`type`:“countdown”, `auctionId`:“a42”]
[`type`:“accept”, `auctionId`:“a42”, `bid`:12, `bidderId`:“bidder1”]

For the management of these entries, the `QueryCoordinator` and a special `Auction-Coordinator` are used as obligatory coordinators. Similar to the `KeyCoordinator`, this user-defined coordinator prevents auctions with identical ids from being created. Additionally, it checks for bid entries that the corresponding auction still exists in the space when the bid is issued and that each bid has at least the value of the starting bid. The coordinator, however, does not check if the bid is higher than the current maximum bid because bidders can withdraw their offers so that lower bids might win too. Therefore, the CoXT only needs to store the id and starting bid for each auction and every action beside the creation of auctions may be invoked in concurrent transactions. The query function

determines if the auction is still active by checking its internal auction list because deleted auctions are also removed from the CoXT by the `onRemove` accountant function. If it cannot be found, a `NOTOK` error must be raised. Otherwise, the function selects a winning bid of an auction according to the following policy: As long as a specified number of distinct bidders are involved in the auction with active bids, the highest bid should be selected. This behavior can be described with the XVSMQL query `type="bid"` | `auctionId=`*`id`* | `sortdown(bid)` | `distinct(buyerId)` | `cnt(`*`k`*`)` | `cnt(1)`, where `k` is the number of distinct bidders required, given as an argument to the query function, which also requires the auction id.

Sellers create auctions by writing auction entries into the container. Then, they can wait until a defined number of different bidders has issued offers by invoking a read operation using an auction selector that includes this number. The exact conditions for the acceptance of a bid can be easily adapted for each seller individually. In this example, the seller application waits until either a minimum number of bidders has issued bids or a maximum wait time has passed. This is achieved with a blocking read operation using the maximum wait time as timeout. If this operation fails, a non-blocking read is invoked where `k` is set to 1 so that there is no restriction on the number of bidders as long as at least one bid exists. Then, a countdown is started for the auction in both cases and a countdown entry is written to the space using a `KeyCoordinator` with the auction id. After a specified amount of time, another non-blocking read operation is called with `k=1`. If the retrieved bid is higher than the previously read one, the countdown is restarted. Otherwise, the auction is closed and the buyer with the highest bid wins. At the end of an auction, the garbage collection is invoked which deletes all entries with `auctionId=`*`id`* via the `QueryCoordinator`, using the same transaction as the most recent `AuctionCoordinator` read operation. In this transaction, also an acceptance entry for the auction is written to the container with the `KeyCoordinator`. The transaction is needed to guarantee that bidders cannot withdraw the bid after they are selected as winners. Also, the update of the auction status with the `KeyCoordinator` and the removal of the auction appear atomically. Alternative semantics for the seller can be implemented without changing the coordinator. E.g., a seller could just wait for a fixed time and then use a non-blocking `AuctionCoordinator` query with `k=1` to get the highest bid, before an optional countdown is started. Arbitrary conditions for the end of an auction can also be specified with special `QueryCoordinator` read operations on all bids of an auction.

Bidders can search for auctions of books with particular title or authors by applying a read operation on the container with the `QueryCoordinator`. Then, they can issue bids on these auctions by writing bid entries to the space. Bids can be cancelled before the given auction ends with a delete operation using the query `type="bid"` | `auctionId=`*`myAuctionId`* | `bidderId=`*`myId`*. A bidder can dynamically react to the

progression of an auction by starting two listener threads. The first listener waits for any bid that is higher than the highest own bid by blockingly reading the container with the combination of an `AuctionCoordinator` query with `k=1` and a query selector with parameter `bidderId` $\neq$ *myId* | `cnt(1)`. Thus, this function allows bidders to wait until someone outbids them, so that they can react by placing another bid. The second listener function waits for status updates on the auction, which can be either countdown or acceptance messages. These can simply be retrieved by a blocking read query with a key selector that uses the `auctionId` as parameter. After the countdown has been started, the listener may add a query selector to the read operation specifying that only acceptance entries should be found. Otherwise the countdown message would be returned constantly.

For this auction scenario, the XVSM model provides high flexibility with minimal effort for the developers of seller and bidder applications. In contrast, a database solution would be less reactive as it would require polling to wait for bids. Moreover, database systems are usually not well suited for ad-hoc coordination because they must be installed and maintained by an administrator, whereas spaces can be created dynamically in a simpler way. Compared to a JavaSpaces implementation, XVSM supports much better coordination mechanisms. Simple template matching does only allow equality checks, thus it is not possible with the classical Linda model to search for a higher bid. Therefore, coordination and business logic have to be mixed in the application, which reduces maintainability, while an XVSM based approach separates coordination from computation.

6 Haskell XVSM Prototype

In the previous sections the formal model of XVSM has been presented. Based on this model a reference implementation in the functional programming language Haskell [19] has been created for various reasons. It should help to detect design flaws in the formal specification, enable extensive tests of the formal model in various use cases and define clear semantics in every situation. Further XVSM implementations can use this Haskell prototype as a reference for the semantic behavior of the XVSM core in ambiguous cases, which should help to create interoperable versions of XVSM on various platforms. The focus of the Haskell prototype is clearly on the semantics of a single XVSM core in highly concurrent situations whereas performance issues and usability are only of second priority, as the intended users are developers of XVSM runtimes and not application programmers.

The use of Haskell, one of the most common functional programming languages nowadays, was a logical choice because the functional programming paradigm [18] tends to map formal specifications more clearly than imperative languages like Java. In this programming style, which is based on lambda calculus introduced by Alonzo Church, the result of a program is computed by evaluating mathematical functions instead of performing a series of actions like in imperative programming. A function's result only depends on its input because the program does not have any inner state. Destructive updates of variables like in imperative languages are not possible and thus side effects, which change the values of other computations and are the cause of many programming errors, are prevented. Instead of loops, recursion must be used for iterations. Higher-order functions enable programmers to use other functions as input or output of their own computations. All these properties allow the programmer to create short and readable code despite the lack of destructive updates.

6.1 Introduction to Haskell

The programming language Haskell, which was originally created in 1990 as a common foundation for research on functional programming, is widely used in research and also in commercial applications in its current standardized form, Haskell 98 [37]. For the XVSM formal model prototype, the Glasgow Haskell Compiler[4] (GHC), which is the de-facto reference implementation of Haskell 98, is used.

As a purely functional programming language, Haskell forbids side effects in its functions. The result of any function depends only on the values of its input parameters and not on the history of prior calls. Haskell is a non-strict language, meaning that expressions need not be evaluated if they are not necessary for the computation of the function result. Therefore, Haskell applies a lazy evaluation strategy which does not

[4] http://www.haskell.org/ghc/

compute the results of expressions until they are needed. A strong static type system is used, so that every variable has a definite immutable type that can be determined at compile time. The types of input parameters and return values are usually explicitly defined by the programmer whereas the types of other variables can be implicitly inferred by the compiler as long as they are not ambiguous. However, Haskell also allows type variables symbolizing arbitrary types, which enables the programmer to create generic functions. Furthermore, type classes are supported that declare a set of functions that must be defined by member types. Type classes are used to support ad-hoc polymorphism so that generic functions can be defined with input variables that need to have a certain function, like e.g. an equality check defined in the type class `Eq`. Haskell supports higher-order functions, i.e. functions can be used as input parameters or return values and stored in data structures like any other variable type. Functions in Haskell are often found in their curried version, which means that instead of taking a list of its arguments as input, a function only takes its first input parameter and returns another function that requires the second argument of the original function and so forth. When all arguments of a function are given, both versions return the same result, but currying enables partial evaluation of a function when not all arguments are available.

```
1 getXTree0 :: XTree -> String -> XTree
2 getXTree0 (Ms []) _ = Nil
3 getXTree0 (Ms (x:xs)) lbl
4     |isLbl lbl x   = getVal x
5     |otherwise     = getXTree0 (Ms xs) lbl
6 ...
```

Listing 3: A simple Haskell function (snippet)

Listing 3 shows a code snippet of a simple function which takes an xtree and a string as arguments and returns a child xtree with the given label or `Nil` if none is found. The first line specifies the signature of the function. Strictly speaking, it is a function that takes an `XTree` and returns another function that takes a `String` and returns an `XTree`. However, informally all but the last element of the signature can be considered as the function's arguments, while the last one is the return value. Lines 2 and 3 show two different definitions for `getXTree0`, depending on the form of the given arguments. In the example snippet, the function is only defined for multisets which are identified by their constructor `Ms`. Empty multisets use the first definition, while for non-empty ones, which can be described as a composition of a head element `x` and the tail `xs`, the second definition applies. This approach is called pattern matching. Any function is executed by using the first function definition that matches the given arguments, which are then bound to variable names that are used in the function's body to compute the result. If an argument is not required for the computation of the result, a wildcard that matches any argument (_) can be used as a pattern. A function can be defined with guards like in

lines 4 and 5. It is evaluated by returning the first right-hand expression for which the Boolean expression of a guard is true. For the example function, this means that if the first property's label is equal to the specified string, the value xtree of this property is returned, otherwise a recursive call to `getXTree0` with the remaining multiset properties as argument continues the search. The helper functions `isLbl` and `getVal` are defined elsewhere.

Other important language features are `where` and `let` clauses within function definitions that allow local variable bindings, a module concept that enables separation of code, the ability to define anonymous functions as arguments for other functions, and a powerful way to generate lists of data via list comprehension. Furthermore, conditional expressions can be formed via case and if-then-else constructs. In contrast to many other languages, the layout in Haskell actually matters, as the indentation determines the scope of blocks. Lines with the same starting column belong to the same block, whereas an expression can be extended to multiple lines by an additional indentation of the extra lines.

A further distinguishing feature of Haskell are monads, which allow the use of side effects in programs without eliminating the language's purely functional character. A monad can be viewed as an abstract data type that allows the programmer to specify a chain of actions in an imperative style, which can be useful for interacting with the user (e.g. via the `IO` monad) or for storing a global state. Monadic functions support a special syntax: A basic block starts with the `do` keyword, followed by a sequence of actions that must be executed. Monadic values can be bound to variables via the bind operator (`<-`) and reversely, monadic values can be created from normal ones via `return`. Exceptions can also be raised within monads with `fail`. The last action of a do-sequence must be an expression that returns a monadic value.

Other advantages of Haskell are the big variety of powerful built-in functions and the extensive standard and non-standard libraries. Especially the built-in generic list functions help to keep the code short and simple. For example, the function `map` applies a function to each element of a list and returns the result list, while `filter` extracts all elements from a list that fulfill a certain predicate. Further information on Haskell and a detailed description of the language's syntax and semantics together with a wide range of tutorials can be found on the official project site[5].

6.2 Implementation

The prototype focuses on a single XVSM core, thus the protocol layer with sender and receiver processes is omitted. The space can be directly accessed by using the synchronous

[5] `http://haskell.org`

embedded API interacting with the runtime's request and response container. For simulation of the XVSM space and its runtime machine, the capabilities of Haskell 98 are not sufficient. Therefore, two non-standard extensions are used. *Concurrent Haskell* [39, 36] offers the possibility to create concurrent threads, which is needed to implement the different runtime processes. Using the primitive `forkIO`, which takes a function of the `IO` monad as an argument, a new process is started within the Haskell runtime while the parent function continues with the next action. A `ThreadId` is returned that can be later used to stop the thread with `killThread`. Another method that is used in the XVSM implementation is `threadDelay`, which forces the own thread to sleep for a specified time. This is used for the timeout processor to periodically poll for expired requests and transactions. Listing 4 shows how the XVSM runtime can be initiated with `startRuntimeThreads` by forking three core processors and one timeout processor. The returned `ThreadIds` can later be used to kill the processes with `stopRuntimeThreads`.

```
1  startRuntimeThreads :: TVar Space -> IO [ThreadId]
2  startRuntimeThreads sref =
3    do xp1 <- forkIO (coreRoutine sref)
4       xp2 <- forkIO (coreRoutine sref)
5       xp3 <- forkIO (coreRoutine sref)
6       tp <- forkIO (tpRoutine sref)
7       return [xp1, xp2, xp3, tp]
8
9  stopRuntimeThreads :: [ThreadId] -> IO ()
10 stopRuntimeThreads threads = mapM_ killThread threads
```

Listing 4: Creation and destruction of runtime threads

In this example, the space is stored in a variable of type `TVar Space`, which is provided by the second used Haskell extension, namely *Software Transactional Memory* (STM) [16, 11, 38]. The STM concept, which was originally proposed in [45], allows to group several actions on a shared data structure into one atomic operation. Instead of locking, an optimistic approach is used to avoid conflicts when concurrent threads access the same variable. The STM monad in Haskell provides mutable variables of type `TVar a`, which can be modified within transactions, where "`a`" stands for an arbitrary type. For the XVSM implementation, STM offers two main features: It guarantees the atomic space access of CAPI-1 and parts of CAPI-2 via the `atomically` primitive and it helps to simulate the blocking behavior of CAPI-B with the `retry` function. With `atomically`, which is used within the `IO` monad, a series of STM actions can be executed within a transaction. Possible actions include the creation of a new mutable space variable with `newTVar` and a read operation `readTVar` that returns the currently stored space from a `TVar` variable. On the retrieved space variable, any non-monadic transformation can be applied, before the thereby changed space is written back to the global `TVar` with `writeTVar`. For simplicity, the XVSM prototype only uses a single `TVar` for the whole space, which is accessed by all internal methods that require direct interaction with the

space. Listing 5 shows a typical usage of STM within the XVSM implementation. The example function takes a space reference as argument and performs an atomic and possibly blocking update operation on it. On the space variable retrieved from the reference, some sort of conditional check is invoked. If it returns `True`, a modified space variable `s'` is created which is then written into the `TVar`, replacing the original space. Otherwise, the `retry` function is invoked, which blocks the thread until another thread changes the transactional variable `sref` and the execution of the current memory transaction can be retried.

```
1 exampleSTM :: TVar Space -> IO ()
2 exampleSTM sref =
3   atomically (do s <- readTVar sref
4                  if doSomeCheck s
5                    then do let s' = updateSomething s
6                            writeTVar sref s'
7                    else retry)
```

Listing 5: Usage of STM

The transactions provided by STM could be applied to simulate the transactions of XVSM, but in the current prototype STM is only used for the synchronized data access. The Haskell implementation should serve as a reference for the formal model described in the previous sections and therefore STM is not used for the higher CAPI layers as it would hide complex parts of the program logic and allow less control on the semantics of transactions in XVSM. Thus, CAPI-1 methods can be called within STM transactions but all functions of higher CAPI layers are only invoked as `IO` actions. The `IO` monad enables an imperative style for these layers which is necessary to comply with the formal model specification in Sect. 4. Purely functional code is still used for the algebraic foundation, the query language and the methods of CAPI-1, as well as for various helper methods.

6.2.1 Architecture

The XVSM prototype is split up into modules that resemble the structure of the formal model. The basic data types are defined in the `Types` module, while the algebraic access to xtrees is defined in `Capi0`. These modules are used by all other program parts. In `Queries`, which is used by the runtime and all lower CAPI layers, the XVSMQL is modeled. CAPI-1 is broken up into three modules: The purely functional methods of `Capi1` are invoked directly by the runtime within STM actions and by the other CAPI layers to issue a query on xtrees that are not located in the space. `Capi1STM` encapsulates the functions of `Capi1` into `atomically` blocks and is used for simple access to the space by `Capi2`, while `Capi1Debug` maps the methods to corresponding versions that can be invoked by user requests via the management API. The `Capi3` module uses the transactional operations of `Capi2` and the generic coordinator functions of `Coordinators` as

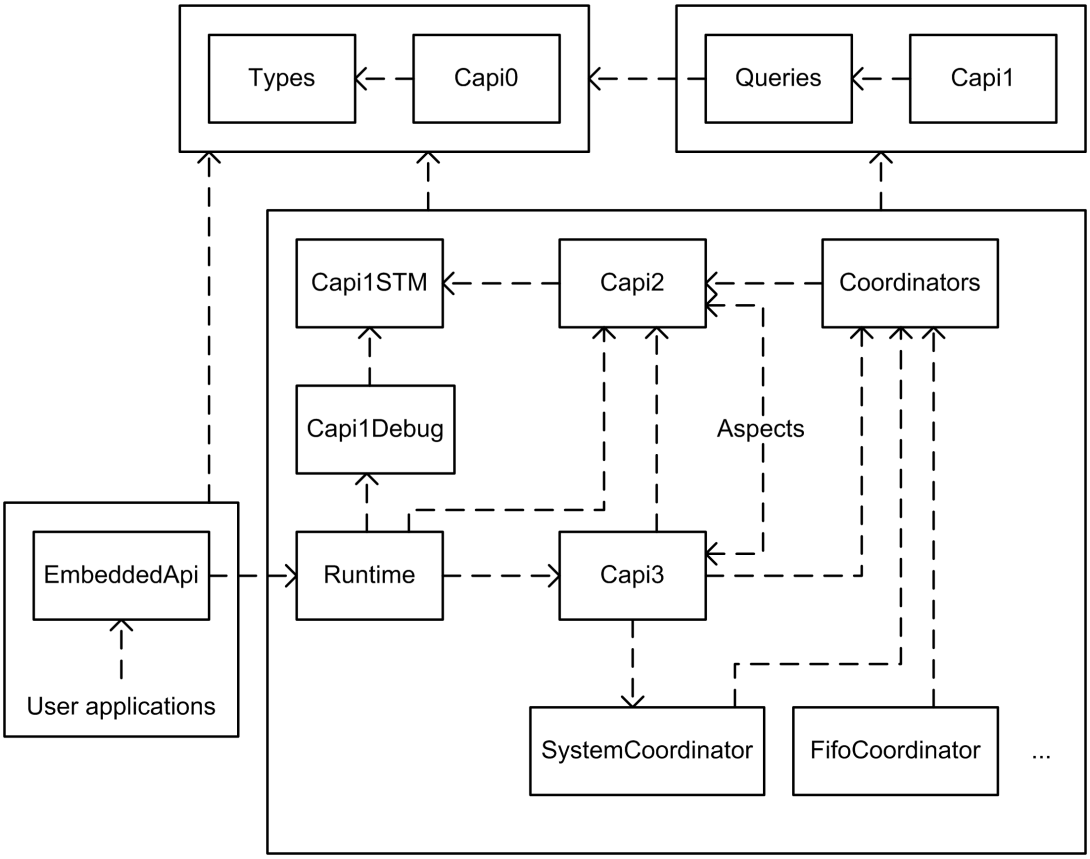

Figure 8: Haskell XVSM prototype architecture

well as the actual coordinator implementations. These predefined coordinator modules also use the `Coordinators` module which provides limited access to CAPI-2. CAPI-4 is implemented via the `Runtime` module that internally uses `Capi1` together with STM to simulate the possibly blocking behavior of CAPI-B. To fulfill user requests, the appropriate operations of `Capi2`, `Capi3` and `Capi1Debug` are called. The predefined coordinator implementations and the dynamically added aspects, which have direct access to `Capi2` and `Capi3`, are also registered here. The `EmbeddedApi` module, which accesses the runtime, finally allows applications to use the XVSM space.

Figure 8 shows the modules and their dependencies. The arrows indicate which modules call functions of other modules directly, whereas several modules are grouped together to simplify the relations. E.g., changes to the `Types` module would require adaptations in all other parts of the implementation because every module depends on the definitions of this module. However, modifications to the runtime may only affect the embedded API and therefore indirectly also user applications.

6.2.2 Basic data access

Listing 6 shows the definition of the `XTree` data type, which is either a multiset (`Ms`), a sequence (`Seq`) or a value. For simplicity, only strings, integers and Boolean values as well as coordinator and aspect definitions are supported. The `Nil` value is used in functions that use xtrees as argument or return value to indicate that the xtree does not exist. To allow comparison between two xtrees, the data type is defined as instance of the type classes `Eq` and `Ord`. Both multisets and sequences are defined as lists of properties, which are themselves tuples of a label string and an xtree. The types `Space`, `Container` and `Entry` are used as synonyms for `XTree` if applicable.

```
1  data XTree = Ms [Property]
2             |Seq [Property]
3             |Str String
4             |Int Integer
5             |Bool Bool
6             |Co Coordinator
7             |Asp Aspect
8             |Nil deriving (Eq, Ord)
9
10 type Property = (String, XTree)
11
12 type Space = XTree
13 type Container = XTree
14 type Entry = XTree
```

Listing 6: XTree data type

Various basic functions are defined that depict the possibilities of the XVSM algebra outlined in Sect. 2 and allow the usage of `XTrees` in the XVSM implementation. With `getXTree0` and `getXTreeByPos0`, a child xtree of a sequence or multiset can be retrieved via its label or index position, respectively, which enables the selection of an xtree via its path. Similarly, the `updateXTree0` and `updateXTreeByPos0` functions allow write access on any child xtree. The `addXTree0` function can be interpreted as a union operator for multisets and as concatenation for sequences, while `deleteXTree0` allows the removal of a child xtree from its parent. With `getSize`, the cardinality of the xtree can be computed. `Capi0` further includes methods to extract the labels and values of properties as well as type checks and conversion functions that are used when an xtree of a specific type is expected, e.g. an integer value. Additionally, several convenience methods are provided that ease the manipulation of multiset and sequence xtrees.

6.2.3 CAPI-1 and query evaluation

As shown in Listing 7, all CAPI-1 functions require an xtree argument specifying the root and a path given as a list of strings. In contrast to the formal model, the return value is not a single xtree. Instead, the result is split into tuples or triples, so that the caller does

not need to extract the needed data from an xtree. The result includes the changed root xtree (except for `read1`), the actual outcome of the operation (except for `writeL1`) and the status xtree that contains the status code and possible error details.

```
writeL1 :: XTree -> [String] -> String -> XTree -> (XTree, XTree)
write1 :: XTree -> [String] -> XTree -> (XTree, String, XTree)
writeBulk1 :: XTree -> [String] -> [XTree] -> (XTree, [String], XTree)
read1 :: XTree -> [String] -> [QueryFunc] -> (XTree, XTree)
take1 :: XTree -> [String] -> [QueryFunc] -> (XTree, XTree, XTree)
```

Listing 7: Signatures of Capi1 functions

The read and take operations use a list of `QueryFunc` objects as parameters. Each of these functions corresponds to an SXQ of the query language described in Sect. 3. Listing 8 shows the definition of the data type as well as examples for query function implementations. A query function can either be a selector with constructor `Sel` or a matchmaker as indicated by `Mm`. A matchmaker is simply a function that returns `True` or `False` for any specified property. E.g., the matchmaker used for testing a specific xtree of a property according to a given predicate can be defined via `qTest`. By specifying the path of the (sub) xtree that should be examined and a function that takes this xtree as an argument and returns a Boolean value, a matchmaker query function is created. By using partly evaluated functions, a wide range of matchmakers can be derived from this definition. For example, the XVSMQL matchmaker `*/author = "Gelernter"` can be expressed with the query function `Mm (qTest ["*", "author"] (== Str "Gelernter"))`. The implementation of `qTest` uses a helper method to retrieve all sub xtrees of the property that match the given path. On these xtrees, the specified comparison predicate is applied using the predefined `any` function, which returns `True` if the comparison predicate is fulfilled by at least one matching xtree. Selector functions use a list of indexed xtrees as argument that represents a sequence or multiset container. The query execution logic requires that each property must be uniquely identifiable within the root xtree, which is achieved with a integer list representing the index positions of each parent xtree in its own parent xtree. Additionally, the full path of the property is needed. Therefore, the container on which the selector function should be applied is represented by a list of triples with index list, path and value as well as a Boolean flag that indicates if the input container is a sequence or a multiset. This data structure is also returned after the selector function has changed it, together with an xtree that contains possible errors. As an example for a selector, the `qCnt` function can be used with an integer argument, e.g. `qCnt 3` corresponds to `cnt(3)`. For any specified positive integer `n`, this function simple takes the first `n` properties and returns the reduced list. If not enough properties are available, an error is returned.

```
data QueryFunc = Sel ([([Integer], [String], XTree)] -> Bool
                      -> ([([Integer], [String], XTree)], Bool, XTree))
               |Mm (Property -> Bool)

qTest :: [String] -> (XTree -> Bool) -> Property -> Bool
qTest compPath pr prop = any pr xts
    where
        xts = getQueryPathXTs (Ms [prop]) compPath

qCnt :: Integer -> [([Integer], [String], XTree)] -> Bool
          -> ([([Integer], [String], XTree)], Bool, XTree)
qCnt cnt ixProps isSeq
    |cnt <= 0                          = ([], isSeq, Str "InvalidArgument")
    |cnt > toInteger (length ixProps)  = ([], isSeq, Str "CountNotMet")
    |otherwise                         = (take (fromInteger cnt) ixProps, isSeq, Nil)
```

Listing 8: Query functions

When using `read1` or `take1`, initially all xtrees in the search path specified by the path argument are retrieved with attached index lists that serve as references. If the list of query functions is empty, these xtrees represent the result. Otherwise, the child xtrees of all these xtrees are aggregated in a single multiset or sequence, which is then used as input for the `queryExecution` method shown in Listing 9. This recursive function applies selector functions directly on the input container structure, while matchmaker functions are invoked via a special `matchmaker` method that executes the given predicate on all available properties. The indices of the result properties are finally used to remove the correct properties in the case of `take1`, while the property path and the value xtree are included in the function's result xtree.

```
queryExecution :: [([Integer], [String], XTree)] -> Bool -> [QueryFunc]
                  -> ([([Integer], [String], XTree)], Bool, XTree)
queryExecution xt isSeq [] = (xt, isSeq, Nil)
queryExecution xt isSeq ((Sel q):qs)
    |exists errorXT   = res
    |otherwise        = queryExecution qResult qIsSeq qs
    where
        res@(qResult, qIsSeq, errorXT) = q xt isSeq
queryExecution xt isSeq ((Mm q):qs)
    |exists errorXT   = (qResult, isSeq, errorXT)
    |otherwise        = queryExecution qResult isSeq qs
    where
        (qResult, errorXT) = matchmaker q xt

matchmaker :: (Property -> Bool) -> [([Integer], [String], XTree)]
              -> ([([Integer], [String], XTree)],  XTree)
matchmaker mmFunc props = (filter (\(_, path, xt) -> mmFunc (last path, xt)) props, Nil)
```

Listing 9: Query execution

6.2.4 CAPI-2

In contrast to `Capi1`, the operations of the `Capi2` module only return one xtree as described in the formal model. The atomic access on the `_locks` and `_txC` container is achieved with the help of STM by setting special lock xtrees on which a blocking take is issued. As these locks are always acquired and released in the same order, they block an operation only for a very limited time and no deadlocks can occur, so this does not contradict the non-blocking behavior defined for CAPI-2 in the formal model.

The transactional operations of `Capi2` use various lock check methods that help to decide if an access to a specific xtree is allowed or not. If these checks succeed, a lock is written to the `_locks` container and corresponding entries are created for the log of the involved operation, as described in Sect. 4.2. The query used for the `canBeRead` function, which uses the `atomicRead1` read method defined in `Capi1STM`, is shown in Listing 10. According to the lock compatibility defined in Table 3, any foreign delete or exclusive read lock on either a parent xtree, the examined xtree itself or a child xtree leads to a denial of the read access. The query on the `_locks` meta container of the given space consists of two matchmaker functions. The first one is a test on the path property of a lock. The `isChildOrParentOrExactPath` predicate compares the path of the xtree that should be read with the path of the lock. If the paths are equal or any of the paths is a parent xtree of the other, the entry is included in the intermediate result that is passed to the second matchmaker. This function checks if the lock is indeed a delete or exclusive read lock and that it is not held by the own operation, which is specified via the `tx` and `op` parameters. If this query returns a non-empty result, the `canBeRead` function indicates an incompatible lock, otherwise this check succeeds.

```
1 atomicRead1 sref ["_locks"] [Mm (qTest ["*", "path"] (isChildOrParentOrExactPath path)),
2                              Mm (qAnd (qOr (qTest ["*", "type"] (== Str "DELETE"))
3                                            (qTest ["*", "type"] (== Str "EXCL")))
4                                       (qOr (qTest ["*", "tx"] (/= Str tx))
5                                            (qTest ["*", "op"] (/= Str op))))]
```

Listing 10: Query if xtree can be read

When committing, the `commitTransaction2` function extracts the logs of all finished operations within the specified transaction and passes them to the `commitLogs` method shown in Listing 11, which invokes the `commitLogEntry` function on all log entries for each of the operation logs. This method retrieves the type and path properties of the entry and then removes the specified inserted lock or deleted property from the space with a take operation, while log entries of type `PROP-INSERTED` are ignored.

```
1 commitLogs :: TVar Space -> [XTree] -> IO ()
2 commitLogs _ [] = return ()
3 commitLogs sref (lg:lgs) = do mapM_ (commitLogEntry sref) (getValues lg)
4                               commitLogs sref lgs
5
```

```
6  commitLogEntry :: TVar Space -> XTree -> IO ()
7  commitLogEntry sref entry = do let entryType = getStringVal (getXTree0 entry "type")
8                                 let path = getStringValues (getXTree0 entry "path")
9                                 case entryType of
10                                  "LOCK-INSERTED"   -> do atomicTake1 sref path []
11                                                          return ()
12                                  "PROP-DELETED"    -> do atomicTake1 sref path []
13                                                          return ()
14                                  _                 -> return ()
```

Listing 11: Committing a transaction

6.2.5 CAPI-3 and coordinators

The functions defined in `Capi3` enable the access on containers with the help of coordinators, as described in Sect. 4.3. Listing 12 shows how the coordinator implementations are incorporated into this module using the example of the `evalSelectorChain` method which is used by `read3`, `take3` and `delete3` to get the query result for a given chain of read selectors. The method is called recursively for every read selector specified by its id (`coordId`) and its arguments (`argsXT`), whereas the current result set stored in the `inEntries` parameter is used to pass the input entries to the next coordinator in the chain. If all read selectors are processed, these entries are returned in the method's result (line 4). The `coordImpls` argument contains a previously retrieved list of coordinator implementations, which is used to find the appropriate query function for the current read selector (lines 7–10). This function is then called and for result status `OK`, the recursion is invoked with the output entries of the current coordinator as input for the next one. If an error occurs, the corresponding status xtree is forwarded instead.

```
1  evalSelectorChain :: TVar Space -> [String] -> [(String, Coordinator)] -> XTree
2                       -> [(String, XTree)] -> XTree -> OperationId -> IO XTree
3  evalSelectorChain sref containerPath coordImpls inEntries [] contextXT opid =
4    return (Ms [("status", Str "ok"), ("result", inEntries)])
5  evalSelectorChain sref containerPath coordImpls inEntries
6                    ((coordId, argsXT):sels) contextXT opid =
7    do let lookupRes = lookup coordId coordImpls
8       if isJust lookupRes
9         then do let coordImpl = fromJust lookupRes
10                let queryFunc = getQueryFunction coordImpl
11                queryXT <- queryFunc argsXT sref coordId containerPath
12                                     inEntries contextXT opid
13                if isOkCapi2 queryXT
14                  then do let resultEntriesXT = getXTree0 queryXT "result"
15                          evalSelectorChain sref containerPath coordImpls
16                                            resultEntriesXT sels contextXT opid
17                  else return queryXT
18        else return (Ms [("status", Str "notok"), ("details", Str "UnknownCoordinator")])
```

Listing 12: Selector chain evaluation

The coordinators themselves do not access the functions of `Capi2` directly to interact with the container. Instead, the `Coordinators` module provides common helper functions that map the functionality of CAPI-2 with the restrictions applicable to coordinator functions. With `readCoXT`, `takeCoXT`, `writeCoXT`, `writeBulkCoXT` and `lockCoXT`, the corresponding `Capi2` functions are invoked on the CoXT of the calling coordinator. Listing 13 shows the `queryEntries` function, which is used within coordinator query functions to select entries from the container. The method adds a matchmaker to the given query that removes entries which are not managed by the calling coordinator (line 7). Additionally, only the entries included in the input may be visible for the query. Therefore, a special selector `qLabelRef` is used that only selects entries with a property label included in the argument list, while preserving the sorting of the entries according to the label list. For the first coordinator of the query chain, the `inEntries` parameter is `Nil`. Therefore, the whole container is visible. Subsequent query functions obtain a list of input entries from the `evalSelectorChain` function. The extracted labels of these entries are then used for the `qLabelRef` selector that is also prepended to the query, which is finally invoked on the container entries via `read2`.

```
1  queryEntries :: TVar Space -> [String] -> XTree -> String
2                  -> [QueryFunc] -> OperationId -> IO XTree
3  queryEntries sref containerPath inEntries coordId query opid =
4    do let filterQuery = if exists inEntries
5                         then [Sel (qLabelRef (getLabels inEntries))]
6                         else []
7       let query' = [Mm (qTest ["*", "coordinators", "*", "id"] (== Str coordId))]
8                    ++ filterQuery ++ query
9       readXT <- read2 sref (containerPath ++ ["entries"]) query' opid
10      if isOkCapi2 readXT
11        then do let entries = getQueryXTree (getXTree0 readXT "result")
12                return (Ms [("status", Str "ok"), ("result", entries)])
13        else return readXT
14
15 qLabelRef :: [String] -> [([Integer], [String], XTree)] -> Bool
16              -> ([([Integer], [String], XTree)], Bool, XTree)
17 qLabelRef lbls ixProps isSeq =
18   (concatMap (\lbl -> filter (\(_, p, _) -> last p == lbl) ixProps) lbls, isSeq, Nil)
```

Listing 13: Container read access for coordinators

Each of the predefined coordinators is specified in an own module. Listing 14 shows the most important parts of the `KeyCoordinator` as an example. As shown in lines 1–3, any coordinator implementation is stored in a 7-tuple including a unique reference as well as the init, accountant and query functions. Basically, the coordinator functions use the parameters specified in the formal model with some minor adaptations: Instead of a single entry parameter, the accountant functions use separate arguments for the entry label, its user data and the appropriate entry CoXT, which is more practical because the coordinators do not have to extract these data manually. The transaction and operation

ids are combined in a single parameter `opid` because the coordinator itself need not distinguish them as they are just passed to the `Capi2` methods via the interface functions provided by the `Coordinators` module. Additionally, a `coordId` is necessary for the accountant and query functions because in Haskell, coordinators cannot be seen as object instances that are able to store their id in a member variable. Instead the `coordId` is used to enable the association of a coordinator function with the corresponding entry and container CoXTs.

```
keyCoordinator :: Coordinator
keyCoordinator = Coord ("KEY", keyInit, keyInsert, keyRemove, keyRead,
                        keyDataReturn, keyQuery)

keyInsert :: WriteAccountantFunc
keyInsert key@(Str _) sref coordId containerPath elbl entry context opid =
  do lockXT <- lockCoXT sref coordId containerPath ["*", "writeLock"] opid
     if isOkCapi2 lockXT
       then do keyCheckXT <- readCoXT sref coordId containerPath []
                                      [Mm (qTest ["*", "key"] (== key))] opid
               if isOkCapi2 keyCheckXT
                 then do let keyCheckResult = getXTree0 keyCheckXT "result"
                         if getSize keyCheckResult > 0
                           then return (Ms [("status", Str "wait"),
                                            ("details", Str "DuplicateKey")])
                           else writeCoXT sref coordId containerPath []
                                  (Ms [("key", key), ("entryRef", Str elbl)]) opid
                 else return keyCheckXT
       else return lockXT

keyRemove :: AccountantFunc
keyRemove sref coordId containerPath entryCoXT elbl entry context opid =
  takeCoXT sref coordId containerPath []
             [Mm (qTest ["*", "entryRef"] (== Str elbl))] opid

keyQuery :: SelectorFunc
keyQuery (Str key) sref coordId containerPath inEntries context opid =
  do keyXT <- readCoXT sref coordId containerPath []
                       [Mm (qTest ["*", "key"] (== Str key)), Sel (qCnt 1)] opid
     if isOkCapi2 keyXT
       then do let keyEntry = (getQueryValues (getXTree0 keyXT "result"))!!0
               let entryRef = getXTree0 keyEntry "entryRef"
               let query = [Mm (qLabel [getStringVal entryRef])]
               queryEntries sref containerPath inEntries coordId query opid
       else return keyXT
```

Listing 14: KeyCoordinator functions

The `keyInsert` function locks the `writeLock` property to avoid concurrent inserts and then checks if the specified key already exists in the CoXT. If a duplicate key is encountered, a result xtree with status code “wait” is returned, which corresponds to status `DELAYABLE` in the formal model. The `keyRemove` accountant simply takes the key entry from the CoXT with given user entry reference, while the `keyQuery` function searches for one entry in the CoXT with given key. If such an entry exists, the entry

reference `entryRef` is extracted from it. The matchmaker query in line 33 finally selects the entry with the found property label from the container.

6.2.6 Runtime machine and API

The runtime machine consists of two main routines: The `coreRoutine`, which can be started in multiple threads concurrently, implements the XVSM core processor, while the `tpRoutine` includes the timeout processor logic. Additionally, a shutdown listener thread is started, which waits for a special token in the space and then kills all runtime threads.

Requests are fetched in the `coreRoutine` with the help of a blocking take operation realized with the STM `retry` primitive. After the timestamps on a request are updated, the corresponding operation is executed and the result is possibly written to the response container, depending on the status code. Finally, the event logic method `rescheduleProcess` is invoked, before the XP loop is restarted with a new request. Listing 15 shows a simplified version of the `execOperation` method, which is responsible for the execution of a transactional operation. In the shown code, the handling of the event logic properties `subOps` and `waitForOp` is omitted and only the segment related to the `read` operation is listed. An operation is started by invoking the `startOperation2` function and retrieving the operation id, which is also added to the context for the usage within aspects. Then, the global and local pre aspects of the operation are invoked (lines 9–10). The `invokeContainerPreAspects` function returns the modified request and context xtree as well as the status of the aspect execution. If the status equals `SKIP`, the operation's post aspects are called immediately. For status `LOCKED`, `DELAYABLE` or `NOTOK`, the status xtree of the failed aspect is stored in the `result` variable. However, if the pre aspects return `OK`, the actual operation is called. The `read3` operation uses the `selectors` argument that is retrieved from the modified request xtree as well as the possibly changed context xtree. If this invocation succeeds, the post aspects can be executed, otherwise the corresponding status xtree is used as result. The result status of the entire operation is finally examined in lines 29–34. If it is `OK`, the sub transaction is committed via `finishOperation2`, otherwise a rollback is issued with `cancelOperation2`.

```
1  execOperation :: TVar Space -> XTree -> String -> String -> XTree -> IO XTree
2  execOperation sref requestXT opName txStr origContextXT =
3    do startResult <- startOperation2 sref txStr opName
4       let opid = getStringVal (getXTree0 startResult "opid")
5       let origContextXT' = addXTree0 origContextXT "opid" (Str opid)
6       result <- if opName == "write" || opName == "read"
7                     || opName == "take" || opName == "delete"
8         then do let cref = getXTree0 requestXT "cref"
9                 (reqXT, contextXT, aspectStatus) <-
10                  invokeContainerPreAspects sref cref opName requestXT origContextXT'
11                case getXTree0 aspectStatus "status" of
12                  Str "skip"    ->
13                    invokeContainerPostAspects sref cref opName reqXT
```

```
                                              (Ms [("status", Str "ok")]) contextXT
              Str "ok"      ->
                do resultXT <- case opName of
                     "read"    ->
                       do let sels = getXTree0 reqXT "selectors"
                          opResult <- read3 sref (getStringVal cref) (getProps sels)
                                                 contextXT (txStr, opid)
                          return opResult
                     ... (code for write, take, delete)
                   if isOkCapi2 resultXT
                     then invokeContainerPostAspects sref cref opName reqXT
                                                     resultXT contextXT
                     else return resultXT
              _             -> return aspectStatus
    else ... (code for container and management operations)
  let resultStatus = getXTree0 result "status"
  if resultStatus == Str "ok"
    then do finishOperation2 sref (txStr, opid)
            return result
    else do cancelOperation2 sref (txStr, opid)
            return result
```

Listing 15: Request execution (simplified)

An example for a post aspect is presented in Listing 16. This simple filter can be registered at the interception points `postRead` and `postTake` to hide xtrees from the user that do not contain a special flag at path `entries/*/data/visible` of the result xtree. These entries are removed from the result by the aspect function and the modified version is returned.

```
postQueryFilter :: AspectFunc
postQueryFilter sref reqXT resultXT contextXT =
  do let entries = getXTree0 resultXT "entries"
     let filteredEntries =
           Ms (filter (\xt -> (getXTree0 (getXTree0 (getVal xt) "data") "visible")
                              == Bool True)
                      (getProps entries))
     let newResultXT = updateXTree0 resultXT "entries" filteredEntries
     return (addXTree0 newResultXT "contextXT" contextXT)
```

Listing 16: Example aspect

If the status of a request result is `LOCKED` or `DELAYABLE`, the implementation of the event processing logic calls the `scheduleWaitingRequest` function for a specific event category, which checks if the request should be rescheduled or put into the waiting queue. Then, the events generated by the request are determined and processed. An event that is generally visible by all transactions can be evaluated by the `invokeGeneralEvent` method. Both of these event logic helper functions are shown in Listing 17. For the given container and category, the `scheduleWaitingRequest` method reads the timestamps of the last event visible only to the specified transaction and of the last generally visible event. The `evalX` methods are thereby convenience operators that raise an exception if the status xtree returned by a `Capi1` function indicates an error, while otherwise the

result of the function is forwarded without the status xtree. If the more recent of these two events has been triggered after the `lastExecutionTime` of the request, this method writes the request into the request container. Otherwise, it is placed into the corresponding `waitingRequests` queue. In the `invokeGeneralEvent` function, the `lastCommittedTime` timestamp of the given container and category is updated with the current event time. Then, waiting requests that are affected by this event are rescheduled. The request that triggers the event must not wake up itself. Therefore the request id is compared within the take query.

```
1  scheduleWaitingRequest :: Space -> XTree -> String -> String -> String
2                            -> Integer -> Integer -> STM Space
3  scheduleWaitingRequest s reqXT c category tx lastExecutionTime eventTime =
4   do utResult <- evalOp (read1 s ["_waitC", c, category, "uncommittedTime", tx] [])
5      let lastTransactionEvent = if getSize utResult == 1
6                                 then getIntVal ((getQueryValues utResult)!!0)
7                                 else 0
8      lctResult <- evalOp (read1 s ["_waitC", c, category, "lastCommittedTime"] [])
9      let lastCommittedEvent = if getSize lctResult == 1
10                             then getIntVal ((getQueryValues lctResult)!!0)
11                             else 0
12     let lastEventTime = max lastTransactionEvent lastCommittedEvent
13     if lastExecutionTime <= lastEventTime
14       then do (newS, _) <- evalWrite (write1 s ["_reqC"] reqXT)
15               return newS
16       else do (newS, _) <-
17                 evalWrite (write1 s ["_waitC", c, category, "waitingRequests"] reqXT)
18               return newS
19
20 invokeGeneralEvent :: Space -> XTree -> String -> String -> Integer -> STM Space
21 invokeGeneralEvent s reqId c category eventTime =
22  do (_, s1) <- evalTake (take1 s ["_waitC", c, category, "lastCommittedTime"] [])
23     s2 <-
24       evalOp (writeL1 s1 ["_waitC", c, category] "lastCommittedTime" (Int eventTime))
25     (wakeupResult, s3) <-
26       evalTake (take1 s2 ["_waitC", c, category, "waitingRequests"]
27                          [Mm (qTest ["*", "lastExecutionTime"] (<= Int eventTime)),
28                           Mm (qTest ["*", "id"] (/= reqId))])
29     let wakeupRequests = getQueryValues wakeupResult
30     (s4, _) <- evalWriteBulk (writeBulk1 s3 ["_reqC"] wakeupRequests)
31     return s4
```

Listing 17: Event logic functions

The XVSM core can be invoked by user applications via the `EmbeddedApi` module that provides functions to create and shutdown the runtime as well as equivalents for all CAPI-4 operations. These functions are implemented in the following way:

1. A request xtree is built using the operation name and the specified arguments.

2. The runtime function `writeRequest` is called with the request xtree as parameter. This method writes the xtree into the request container and returns a unique id.

3. This id is then used as an argument to `waitForResponse`, which performs a blocking take query on the response container and waits for the response with the specified id, using the `retry` function of STM.

4. The operation result is extracted from the response xtree and returned to the user application.

The runtime also provides various hooks that can be used for internal analysis and debugging. At any time, debug requests can be issued using the management API functions of `Capi1Debug`, which allows arbitrary access to user and meta containers. These requests are handled like normal CAPI-4 operations but they do not trigger any event processing. Additionally, special print commands are available that allow the output of specific parts of the meta model to the console. Finally, all important runtime actions are always stored directly to the space in a special runtime log meta container. Thus, it is possible to trace a request while it is executed by the runtime machine.

6.3 Example

As an example for the usage of the Haskell XVSM prototype via the embedded API, the implementation of the scenario described in Sect. 5 is explained here. The `AuctionCoordinator` accountant functions are implemented similar to the `KeyCoordinator` in case that the corresponding entry is an auction entry. Otherwise the `onInsert` accountant only checks for bid and countdown entries if the related auction is still valid and, for bids, if it has at least the value of the starting bid. The query function checks if the auction exists before it retrieves the best bid by using the `readEntries` method of the `Coordinators` module with a query corresponding to the specified coordinator policy.

To initialize the scenario, a new container is created with obligatory coordinators for auction and query access as well as an optional `KeyCoordinator`. The reference of this container (`cref`), which also includes the space reference `sref`, is then passed to the methods that implement the seller and bidder logic. Container operations are called with the `EmbeddedApi` functions `writeEntries`, `readEntries` and `deleteEntries`, which invoke the corresponding CAPI-4 operations via the space runtime. These methods require a container reference and read selectors or entries with write selectors, respectively, as well as a possibly empty transaction id, a timeout parameter and the user context, which is always empty in this example. The read selector of the `QueryCoordinator` is currently represented by a special structured xtree format instead of a simple XVSMQL string. This eases the parsing of the query in the coordinator's query function at the cost of readability. The `createTransaction` method requires the space reference, the transaction timeout and the context, whereas `commitTransaction` uses the space reference, the

transaction id and the context as parameters. Listing 18 shows the functions available for a seller application.

```
startAuction :: ContainerRef -> String -> String -> XTree -> Integer -> IO ()
startAuction cref@(sref, _) seller id book startingBid =
  do let auctionEntry = Ms [("type", Str "auction"), ("auctionId", Str id),
                            ("startingBid", Int startingBid), ("seller", Str seller),
                            ("book", book)]
     writeEntries cref [(auctionEntry, [])] Nil tryOnce Nil

endAuction :: ContainerRef -> String -> Integer -> Integer -> Int -> IO XTree
endAuction cref@(sref, _) auctionId k maxWaitTime countdownTime =
  do es <- readEntries cref [("auction", Ms [("auctionId", Str auctionId),
                                             ("distinctBidders", Int k)])]
                       Nil maxWaitTime Nil
       `catch` (\e -> readEntries cref [("auction", Ms [("auctionId", Str auctionId),
                                                        ("distinctBidders", Int 1)])]
                                  Nil tryOnce Nil)
     let cntdwnEntry = Ms [("type", Str "countdown"), ("auctionId", Str auctionId)]
     writeEntries cref [(cntdwnEntry, [("key", Str auctionId)])] Nil tryOnce Nil
     auctionCountdown cref auctionId (es!!0) countdownTime

auctionCountdown :: ContainerRef -> String -> XTree -> Int -> IO XTree
auctionCountdown cref@(sref, _) auctionId maxBidEntry countdownTime =
  do threadDelay (countdownTime * 1000000)
     tx <- createTransaction sref 10 Nil
     es <- readEntries cref [("auction", Ms [("auctionId", Str auctionId),
                                             ("distinctBidders", Int 1)])]
                       tx tryOnce Nil
     let newBidEntry = es!!0
     let oldBid = getIntVal (getXTree0 (getXTree0 maxBidEntry "data") "bid")
     let newBid = getIntVal (getXTree0 (getXTree0 newBidEntry "data") "bid")
     if newBid > oldBid
       then do commitTransaction sref tx Nil
               auctionCountdown cref auctionId newBidEntry countdownTime
       else do let winnerData = getXTree0 maxBidEntry "data"
               let winnerId = getXTree0 winnerData "bidderId"
               let winnerBid = getXTree0 winnerData "bid"
               let acceptanceEntry =
                     Ms [("type", Str "accept"), ("auctionId", Str auctionId),
                         ("bid", winnerBid), ("bidderId", winnerId)]
               deleteEntries cref [("query",
                                    Seq [("", Ms [("func", Str "eq"),
                                                  ("args",
                                                   Seq [("",
                                                         Seq [("", Str "auctionId")]),
                                                        ("", Str auctionId)])])])]
                                  tx tryOnce Nil
               writeEntries cref [(acceptanceEntry, [("key", Str auctionId)])]
                                 tx tryOnce Nil
               commitTransaction sref tx Nil
               return maxBidEntry
```

Listing 18: Auction seller logic

With `startAuction`, a new auction is created and written to the container, whereas `endAuction` determines the winner of the auction according to the policy described in

Sect. 5. The first `readEntries` call waits for enough distinct bidders and returns the best bid entry. If, however, a timeout occurs, an exception is raised by the API, which is handled in the `catch` block by issuing a second non-blocking query without a required number of bidders. If a valid bid is found, the countdown is started by writing a countdown entry to the container followed by a call to the `auctionCountdown` method, which sleeps for a specified amount of time and then checks within a transaction if a higher bid has been placed for this auction. If not, the auction is closed and the countdown message is replaced by an acceptance entry, using the same transaction. Otherwise, the countdown continues with the recursive call to `auctionCountdown`.

The functions that are used by bidder applications are shown in Listing 19. The `searchBookAuction` method enables a user to find an auction with a given title using the query `book/title=myTitle`. With `makeBid`, a new bid is written to the auction container, while `cancelBid` deletes all entries of the bidder for a given auction. The `waitForHigherBid` function implements the listener function which waits for a bid higher than the own bid, whereas the `auctionNotification` listener returns when any countdown or acceptance messages are available for the specified auction. The `waitForAccept` method can be invoked after the countdown for an auction has started to wait for the acceptance message.

```
1  searchBookAuction :: ContainerRef -> String -> IO [XTree]
2  searchBookAuction cref title =
3    do es <- readEntries cref [("query", Seq [("", Ms [("func", Str "eq"),
4                                                       ("args",
5                                                        Seq [("",
6                                                              Seq [("", Str "book"),
7                                                                   ("", Str "title")]),
8                                                             ("", Str title)])])])]
9                            Nil tryOnce Nil
10       let auctionData = map (\e -> getXTree0 e "data") es
11       return auctionData
12
13 makeBid :: ContainerRef -> String -> String -> Integer -> IO ()
14 makeBid cref auctionId bidder bid =
15   do let bidEntry = Ms [("type", Str "bid"), ("auctionId", Str auctionId),
16                         ("bid", Int bid), ("bidderId", Str bidder)]
17      writeEntries cref [(bidEntry, [])] Nil tryOnce Nil
18
19 cancelBids :: ContainerRef -> String -> String -> IO ()
20 cancelBids cref auctionId bidder =
21   do deleteEntries cref [("query", Seq [("", Ms [("func", Str "eq"),
22                                                  ("args",
23                                                   Seq [("", Seq [("", Str "type")]),
24                                                                  ("", Str "bid")])]),
25                                         ("", Ms [("func", Str "eq"),
26                                                  ("args",
27                                                   Seq [("",
28                                                         Seq [("", Str "auctionId")]),
29                                                              ("", Str auctionId)])]),
30                                         ("", Ms [("func", Str "eq"),
```

```
31                                                  ("args",
32                                                   Seq [("", Seq [("", Str "bidderId")]),
33                                                                 ("", Str bidder)])])])]
34                      Nil tryOnce Nil
35
36 waitForHigherBid :: ContainerRef -> String -> String -> Integer -> IO Integer
37 waitForHigherBid cref auctionId bidder maxWaitTime =
38   do es <- readEntries cref [("auction", Ms [("auctionId", Str auctionId),
39                                              ("distinctBidders", Int 1)]),
40                              ("query", Seq [("", Ms [("func", Str "neq"),
41                                                      ("args",
42                                                       Seq [("",
43                                                             Seq [("", Str "bidderId")]),
44                                                            ("", Str bidder)])]),
45                                             ("", Ms [("func", Str "cnt"),
46                                                      ("args", Seq [("", Int 1)])])])]
47              Nil maxWaitTime Nil
48      let higherBid = getXTree0 (getXTree0 (es!!0) "data") "bid"
49      return (getIntVal higherBid)
50
51 auctionNotification :: ContainerRef -> String -> Integer -> IO XTree
52 auctionNotification cref auctionId maxWaitTime =
53   do notif <- readEntries cref [("key", Str auctionId)] Nil maxWaitTime Nil
54      let notifData = getXTree0 (notif!!0) "data"
55      return notifData
56
57 waitForAccept :: ContainerRef -> String -> Integer -> IO XTree
58 waitForAccept cref auctionId maxWaitTime =
59   do accept <- readEntries cref [("key", Str auctionId),
60                                  ("query", Seq [("", Ms [("func", Str "eq"),
61                                                          ("args",
62                                                           Seq [("",
63                                                                 Seq [("", Str "type")]),
64                                                                ("", Str "accept")])]),
65                                                 ("", Ms [("func", Str "cnt"),
66                                                          ("args",
67                                                           Seq [("", Int 1)])])])]
68                               Nil maxWaitTime Nil
69      let acceptData = getXTree0 (accept!!0) "data"
70      return acceptData
```

Listing 19: Auction bidder logic

The presented functions enable the implementation of interactive seller and buyer applications. A seller may have to enter information about the book and the starting bid as well as the required configuration parameters for the acceptance condition. When the user confirms these data, the application forks a new thread and calls `startAuction` followed by `endAuction`. When this function returns, the user is informed about the winning bidder. The bidder application may provide a search interface that calls `searchBookAuction` or similar methods. By calling the `waitForHigherBid` method with a `maxWaitTime` of 0, the currently best bid can be found. If this invocation returns an error because no bids exist for the auction, the starting bid indicates the minimum bid. The user can issue a bid with `makeBid` and fork listener threads for `waitForHigherBid` and

`auctionNotification` so that an immediate reaction to a future event is possible. When a countdown event occurs, the application can call `waitForAccept` to enable an immediate notification on the auction's end. The bidding process could also be automated. E.g., the bidder application could automatically bid for an auction when a countdown message occurs and then outbid other bidders until either the auction is won or a previously specified price limit is reached. By using parallel threads, sellers and bidders are able to process different auctions concurrently.

6.4 Results

The implementation of the formal model in Haskell proves that the specification is well-defined and accurate. Extensive tests with the prototype have shown that the XVSM core behaves as expected and that user applications and additional core features can be implemented with reasonable overhead. As far as possible, the XVSM prototype depicts the semantics defined in Sect. 4. However, some adaptations have been necessary to reflect the functional character of the programming language. The implementation only uses a single global variable that stores the entire space, but apart from this STM value no objects or references exist. In object-oriented programming languages, space elements like containers, transaction or coordinators could be passed to CAPI methods by reference, which enables them to manipulate the data structures directly. In Haskell, however, all parameters beside the space reference are given by value, which means that any time a function needs to change an xtree within the space, it must know the exact path of this entity and then traverse the entire space to access the data structure. Therefore, the XVSM prototype is slower than comparable imperative or object-oriented approaches. However, the purpose of the Haskell XVSM prototype is not to provide an efficient space implementation that is used by user applications. Instead, it should serve as a reference for the design of actual XVSM implementations and thus simplicity is given precedence over performance. The functional approach helps to understand the exact behavior of the space core just by reading the code instead of having to run extensive tests because side effects, which might lead to unexpected behavior as in object-oriented languages, are prevented. Also, many parts of the prototype are designed in a much simpler way than any object-oriented language would allow. E.g., the query language can be implemented with minimal amount of code due to the combination of higher-order functions.

During the implementation, a lot of open questions have occurred that have forced design decisions and thus led to a more precise definition of the formal model, like, for instance, issues regarding the locking behavior or race conditions within the runtime. Also, the requirement of an effective memory management has been discovered, as the xtrees in the XVSM meta model grow very rapidly at runtime, especially due to extensive

data on transactions and events. Therefore, an intelligent garbage collection mechanism is required and efficient data structures must be used that allow fast access to meta data with minimal memory consumption. Future changes to the XVSM formal model should also be applied to the prototype, so that the reference for XVSM core developers is kept up-to-date. Additionally, the prototype can be extended with modules representing the XVSMP layer as soon as a detailed specification of the protocol exists, which would allow to test the network behavior of the space. On top of this layer, a Java API is planned that would allow to compare the behavior of the XVSM specification with an actual Java implementation of the middleware core where the same test classes can be used.

7 Future Work

Based on the presented XVSM formal model and its Haskell implementation, a new Java implementation of the XVSM middleware is currently in development. Compared to the actual MozartSpaces version [44, 42], the general performance and scalability will be optimized and several improvements inspired by the formal model are planned:

- Users will be able to issue XVSMQL queries on entries, which enables more flexible coordination.
- With the enhanced event processing logic, waiting requests will be rescheduled in a more efficient way, while race conditions within the runtime are prevented.
- The core implementation will consist of several modules that are easily exchangeable, which corresponds to the CAPI layer architecture of the formal model.
- Application developers should be able to develop own coordinators in a simpler way.
- The direct access to the standardized XVSM meta model via the management API allows for better logging and debugging capabilities.

Nevertheless, the work on the formal model is far from being finished and it will be constantly improved to factor in requirements of new use cases. Thus, the specification of all CAPI layers will be revised and adapted if necessary. The ultimate goal is to achieve certification for the XVSM technology, so that it could be established as an accepted industrial standard for coordination middleware in the future. One possible research field that requires unambiguously specified and verified semantics is the usage of XVSM for embedded systems and robots, as unexpected behavior of the middleware may lead to costly hardware defects or safety risks. To enable extensive analysis and correctness proofs for important parts of the middleware, the presented specification should be modeled in a more formal way by using some sort of process calculus or logic-based methods. Further important issues subject to future research work are outlined in the following:

- **Fairness:** If several requests are waiting for the same entry of a container, the current runtime model does not specify which operation may execute first and possibly remove the entry after it has been written to the space. Thus, other concurrent operations may not be able to read the entry even if they are waiting longer than the successful operation. A fair runtime must ensure that an older request is always given precedence over a newer one if both are able to execute. This, however, might lead to reduced concurrency.

- **Deadlock detection:** If two concurrent transactions hold exclusive locks that are also required by operations of the other transaction, a deadlock occurs. Currently, the corresponding operations are blocked until one of the transactions times out and is therefore rollbacked. However, a more efficient mechanism could be designed that detects such conflicts immediately.

- **Aspect semantics:** The concrete semantics of aspects need to be specified, especially what kind of actions an aspect may perform. It must also be determined how aspects may access remote XVSM cores. A scripting language will be defined that allows the cross-platform usage of aspects on local and remote spaces. The aspect control commands that enable the user to add and remove aspects could also gain an additional transaction parameter, so that aspects can be managed in a transactionally safe way. Also, aspects that are invoked by the runtime in an asynchronous way could be defined.

- **Isolation levels:** For transactions, additional isolation levels like "read committed" could be supported in the future to enable alternative semantics for certain operations.

- **Shift operation:** Similar to the write operation, a shift method can be defined which writes entries to the space but does not block if the container is full or another coordinator constraint is violated. Instead, all conflicting entries of the container are removed before the given entry is written. It must be specified clearly how the coordinator logic chooses the entries that have to be deleted.

- **XVSMP specification:** The protocol layer, which is responsible for the communication with remote XVSM APIs and runtimes, as well as the used XML communication protocol must be adapted to enable the interaction of different XVSM implementations.

- **Profile definitions:** Several important features of XVSM that are not directly included in the core, like notifications, replication, persistency and security can be specified.

8 Conclusion

Complex distributed applications can be designed in an efficient way by applying space-based computing middleware. The XVSM technology follows this paradigm and enables ad-hoc collaboration between distributed and possibly inhomogeneous peers using a flexible and extensible coordination model and a language independent protocol. In XVSM, peers may write user data into shared containers and access information according to one of many possible coordination laws, which are defined by so-called coordinators. The space further supports blocking behavior, transactions and aspects that enable application developers to enrich space operations with additional semantics. Although a lot of previous work on this middleware was published in the past [44, 42, 43, 22], a formal model of the space's behavior has not been defined yet. Such a formal foundation is required to enable the interoperability of different XVSM implementations and to verify the correct behavior of the system in critical scenarios. Therefore, this thesis provides a specification of the middleware's semantics.

In this model, the shared space that stores application data is represented by a nested data structure called xtree. The functionality of the XVSM core can be bootstrapped with own mechanisms as all meta data needed for the realization of the runtime machine can be stored in the space itself, thus forming the XVSM meta model. The architecture of the middleware is described via several Core API layers. The basic data access operations of CAPI-1 make use of a special extensible query language called XVSMQL that applies a filter chain of selector and matchmaker functions on the content of an xtree. While CAPI-2 realizes transactions, CAPI-3 introduces the container concept and specifies the behavior of coordinators, which decide how entries can be written and queried within a container. The runtime machine, which executes CAPI-4 requests from local and remote peers, reschedules blocked requests when corresponding events occur in the space and it is also responsible for the invocation of user-defined aspects.

As a proof of concept, a prototype of this XVSM specification has been built in the functional programming language Haskell, which serves developers of XVSM core implementations as a reference. This prototype shows that the specification is accurate and enables fast testing of the behavior of XVSM according to arbitrary use cases. As an example, a simple auction scenario has been presented that utilizes many of the middleware features described in this thesis.

References

[1] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, 2004.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[3] M. Bravetti, R. Gorrieri, R. Lucchi, and G. Zavattaro. On the expressiveness of probabilistic and prioritized data-retrieval in Linda. *Electr. Notes Theor. Comput. Sci.*, 128(5):39–53, 2005.

[4] M. Bravetti, R. Gorrieri, R. Lucchi, and G. Zavattaro. Quantitative information in the tuple space coordination model. *Theor. Comput. Sci.*, 346(1):28–57, 2005.

[5] N. Busi, R. Gorrieri, and G. Zavattaro. On the semantics of JavaSpaces. In Scott F. Smith and Carolyn L. Talcott, editors, *FMOODS*, volume 177 of *IFIP Conference Proceedings*, pages 3–19. Kluwer, 2000.

[6] N. Busi, R. Gorrieri, and G. Zavattaro. Process calculi for coordination: From Linda to JavaSpaces. In *AMAST '00: Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology*, pages 198–212, London, UK, 2000. Springer-Verlag.

[7] P. Ciancarini, M. Mazza, and L. Pazzaglia. A logic for a coordination model with multiple spaces. *Sci. Comput. Program.*, 31(2-3):231–261, 1998.

[8] J. Clark and S. DeRose. XML path language (XPath) 1.0. W3C recommendation. *World Wide Web Consortium, http://www. w3. org/TR/xpath*, 1999.

[9] U. Cohen. Inside GigaSpaces XAP. Technical White Paper, GigaSpaces Technologies, 2009.

[10] S. Craß, E. Kühn, and G. Salzer. Algebraic foundation of a data model for an extensible space-based collaboration protocol. In *Thirteenth International Database Engineering & Applications Symposium (IDEAS)*, 2009.

[11] A. Discolo, T. Harris, S. Marlow, S. L. Peyton Jones, and S. Singh. Lock free data structures using STM in Haskell. In Masami Hagiya and Philip Wadler, editors, *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2006.

[12] E. Freeman, K. Arnold, and S. Hupfer. *JavaSpaces Principles, Patterns, and Practice.* Addison-Wesley Longman Ltd., Essex, UK, 1999.

[13] D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[14] D. Gelernter and L. D. Zuck. On what Linda is: Formal description of Linda as a reactive system. In David Garlan and Daniel Le Métayer, editors, *COORDINATION*, volume 1282 of *Lecture Notes in Computer Science*, pages 187–204. Springer, 1997.

[15] T. Haerder and A. Reuter. *Principles of transaction-oriented database recovery*, volume 15. ACM, New York, NY, USA, 1983.

[16] T. Harris, S. Marlow, S. L. Peyton Jones, and M. Herlihy. Composable memory transactions. In Keshav Pingali, Katherine A. Yelick, and Andrew S. Grimshaw, editors, *PPOPP*, pages 48–60. ACM, 2005.

[17] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[18] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[19] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–55. ACM, 2007.

[20] ISO. *ISO/IEC 9075:1992, Database Language SQL*. International Organization for Standardization, 1992.

[21] ISO. *ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF*. International Organization for Standardization, 1996.

[22] M. Karolus. Design and implementation of XcoSpaces, the .Net reference implementation of XVSM – coordination, transactions and communication. Master's thesis, TU Vienna, Institute of Computer Languages, December 2009.

[23] E. Kühn. Fault-tolerance for communicating multidatabase transactions. In *Proceedings of the 27th Hawaii International Conference on System Sciences (HICSS), Wailea, Maui, Hawaii. ACM, IEEE.*, volume 2, pages 323–332, January 1994.

[24] E. Kühn, R. Mordinyi, L. Keszthelyi, and C. Schreiber. Introducing the concept of customizable structured spaces for agent coordination in the production automation domain. *The 8th International Conference on Autonomous Agents and Multiagent Systems*, 2009.

[25] E. Kühn, R. Mordinyi, M. Lang, and A. Selimovic. Towards zero-delay recovery of agents in production automation systems. *2009 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IAT*, 2009.

[26] E. Kühn, R. Mordinyi, and C. Schreiber. An extensible space-based coordination approach for modeling complex patterns in large systems. *3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Special Track on Formal Methods for Analysing and Verifying Very Large Systems*, 2008.

[27] E. Kühn, J. Riemer, and G. Joskowicz. XVSM (eXtensible Virtual Shared Memory) architecture and application. Technical report, Space-Based Computing Group, Institute of Computer Languages, Vienna University of Technology, November 2005.

[28] E. Kühn, J. Riemer, R. Mordinyi, and L. Lechner. Integration of XVSM spaces with the web to meet the challenging interaction demands in pervasive scenarios. *Ubiquitous Computing And Communication Journal (UbiCC), special issue on "Coordination in Pervasive Environments"*, 3, 2008.

[29] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[30] A. Marek. Design and implementation of TinySpaces, the .NET Micro Framework based implementation of XVSM for embedded systems. Master's thesis, TU Vienna, Institute of Computer Languages (in preparation), 2010.

[31] E. J. Mata, P. Álvarez, J. A. Bañares, and J. Rubio. Formal modelling of a coordination system: From practice to theory, and back again. In Gregory M. P. O'Hare, Alessandro Ricci, Michael J. O'Grady, and Oguz Dikenelli, editors, *ESAW*, volume 4457 of *Lecture Notes in Computer Science*, pages 229–244. Springer, 2006.

[32] R. Milner. The polyadic pi-calculus: a tutorial. Technical report, Logic and Algebra of Specification, 1991.

[33] R. Mordinyi, E. Kühn, and A. Schatten. Space-based architectures as abstraction layer for distributed business applications. *Accepted for International Complex, Intelligent and Software Intensive Systems Conference (CISIS)*, 2010.

[34] M. Murth and E. Kühn. Knowledge-based coordination with a reliable semantic subscription mechanism. In Sung Y. Shin and Sascha Ossowski, editors, *SAC*, pages 1374–1380. ACM, 2009.

[35] R. De Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: a kernel language for agents interaction and mobility. *Software Engineering, IEEE Transactions on*, 24(5):315–330, May 1998.

[36] S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, pages 47–96. Press, 2002.

[37] S. Peyton Jones. *Haskell 98 Language and Libraries: The revised report*. Cambridge University Press, 2003.

[38] S. Peyton Jones. Beautiful concurrency. In Andy Oram and Greg Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly and Associates, 2007.

[39] S. L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL*, pages 295–308, 1996.

[40] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[41] P. Poizat, J.-C. Royer, and G. Salaün. Formal methods for component description, coordination and adaptation. In *In WCAT '2004 - Int. Workshop on Coordination and Adaptation Techniques for Software Entities*, pages 84–688, 2004.

[42] M. Pröstler. Design and implementation of MozartSpaces, the Java reference implementation of XVSM - timeout handling, notifications and aspects. Master's thesis, TU Vienna, Institute of Computer Languages, 2008.

[43] T. Scheller. Design and implementation of XcoSpaces, the .Net reference implementation of XVSM – core architecture and aspects. Master's thesis, TU Vienna, Institute of Computer Languages, September 2008.

[44] C. Schreiber. Design and implementation of MozartSpaces, the Java reference implementation of XVSM - custom coordinators, transactions and XML protocol. Master's thesis, TU Vienna, Institute of Computer Languages, September 2008.

[45] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of the 14th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 204–213, 1995.

[46] Sun Microsystems. RPC: Remote Procedure Call protocol specification: Version 2. RFC 1057 (Informational), 1988.

[47] Sun Microsystems. JavaSpaces specification, 1999.

[48] Sun Microsystems. Java Message Service (the Sun Java Message Service (JMS) 1.1 specification), 2002.

[49] A. Wollrath, R. Riggs, J. Waldo, and Sun Microsystems Inc. A distributed object model for the Java system. *USENIX Computing Systems*, 9, 1996.

A XVSM meta model specification

In the following, the XVSM meta model is specified in EBNF syntax [21]. All mandatory properties are defined, whereas additional properties might be added by implementations to extend the functionality or to ease computations. In the following rules, basic value types that are not defined here are italicized.

Xtree data structure:

```
xtree = xMultiset | xSequence | xValue;

xMultiset = "[]" | "[" properties "]";

xSequence = "⟨⟩" | "⟨" properties "⟩";

xValue = string | integer | float | bool (* or any other object *);

properties = property {"," property};

property = label ":" xtree | label | xMultiset | xSequence;

label = string;
```

XVSM spaces:

```
universe = "[" "spaces" ":" xSpaces {"," property} "]";

xSpaces = "[ ]" | "[" spaceProperty {"," spaceProperty} "]";

spaceProperty = spaceURI ":" space;

spaceURI = label;

space = "[" "URI" ":" spaceURI ","
            "containers" ":" xContainers ","
            "coordinatorDefs" ":" xCoordDefs ","
            "aspectDefs" ":" xAspectDefs ","
            "_reqC" ":" xRequestC ","
            "_respC" ":" xResponseC ","
            "_waitC" ":" xWaitC ","
            "_txTimeoutX" ":" xTxTimeoutC ","
            "_txC" ":" xTransactions ","
            "_locks" ":" xLocks
            {"," property} "]";
```

User containers:

```
xContainers = "[ ]" | "[" containerProperty {"," containerProperty} "]";

containerProperty = cref ":" container;

cref = label;

container = "[" ["name" ":" string ","]
                "entries" ":" xEntries ","
                "coordinators" ":" xCoords ","
                "coxts" ":" xCoxts
                {"," property} "]";

xEntries = "[ ]" | "[" entryProperty {"," entryProperty} "]";

entryProperty = label ":" entry;

entry = "[" "data" ":" xtree ","
            "coordinators" ":" xEntryCoords ","
            "coxts" ":" xEntryCoxts
            {"," property} "]";

xEntryCoords = "[ ]" | "[" entryCoordProperty {"," entryCoordProperty} "]";

entryCoordProperty = label ":" entryCoordRegistration;

entryCoordRegistration = "[" "id" ":" coordId {"," property} "]";

coordId = string;

xEntryCoxts = "[ ]" | "[" entryCoxtProperty {"," entryCoxtProperty} "]";

entryCoxtProperty = label ":" "[" coordId ":" xtree "]";

xCoords = "[" coordProperty {"," coordProperty} "]";

coordProperty = label ":" coordRegistration;

coordRegistration = "[" "id" ":" coordId "," "coord" ":" coordinator ","
                        "obligatory" ":" bool {"," property} "]";

xCoxts = "[ ]" | "[" coxtProperty {"," coxtProperty} "]";

coxtProperty = coordId ":" xtree;
```

Coordinator and aspect definitions:

```
xCoordDefs = "[" "SYSTEM" ":" SystemCoordinator ","
               "FIFO" ":" FifoCoordinator ","
               "KEY" ":" KeyCoordinator ","
               "LABEL" ":" LabelCoordinator ","
               "LINDA" ":" LindaCoordinator ","
               "QUERY" ":" QueryCoordinator ","
               "VECTOR" ":" VectorCoordinator
               {"," coordName ":" Coordinator} {"," property} "]";

coordName = label;

xAspectDefs = "[" "read" ":" cAspectDefs ","
                "take" ":" cAspectDefs ","
                "delete" ":" cAspectDefs ","
                "write" ":" cAspectDefs ","
                "createTransaction" ":" aspectDefs ","
                "commitTransaction" ":" aspectDefs ","
                "rollbackTransaction" ":" aspectDefs ","
                "createContainer" ":" aspectDefs ","
                "destroyContainer" ":" aspectDefs ","
                "lookupContainer" ":" aspectDefs ","
                "setContainerLock" ":" aspectDefs ","
                "addAspect" ":" aspectDefs ","
                "removeAspect" ":" aspectDefs {"," property} "]";

cAspectDef = "[" "pre" ":" cAspectList "," "post" ":" cAspectList "]";

cAspectList = "⟨ ⟩" | "⟨ cAspectProp {"," cAspectProp} "⟩"";

cAspectProp = "[" "id" ":" aspectId "," "global" ":" Bool ","
                "aspect" ":" aspectCode
                ["," "cref" ":" cref] {"," property} "]";

aspectId = label;

aspectDef = "[" "pre" ":" aspectList "," "post" ":" aspectList "]";

aspectList = "⟨ ⟩" | "⟨ aspectProp {"," aspectProp} "⟩"";

aspectProp = "[" "id" ":" aspectId ","
               "aspect" ":" aspectCode {"," property} "]";
```

Runtime meta containers:

```
xRequestC = "⟨ ⟩" | "⟨" request {"," request} "⟩";

request = "[" "id" ":" requestId "," "op" ":" string
            ["," "context" ":" xtree] ["," "spaceURI" ":" spaceURI]
            ["," "callerURI" ":" spaceURI]
            ["," "timestamp" ":" time "," "lastExecutionTime" ":" time
            ["," "expireTime" ":" time]] {"," property} "]";

requestId = string;

xResponseC = "[ ]" | "[" response {"," response} "]";

response = "[" "id" ":" requestId "," "result" ":" resultXT
             ["," "callerURI" ":" spaceURI] "]"

resultXT = "[" "status" ":" string ["," "details" ":" xtree]
              ["," "subOps" ":" eventXTs] ["," "waitForOp" ":" eventXT]
              {"," property}"]";

eventXTs = "⟨ ⟩" | "⟨" eventXT {"," eventXT} "⟩";

eventXT = "[" "op" ":" string ["," "cref" ":" cref]
            ["," "tx" ":" txRef] "]";

txRef = label;

xWaitC = "[ ]" | "[" cWaitC {"," cWaitC} "]";

cWaitC = cref ":" "[" "insert" ":" waitC "," "remove" ":" waitC ","
                    "st_unlock" ":" waitC "," "lt_unlock" ":" waitC "]";

waitC = "[" "lastCommittedTime" ":" time "," "uncommittedTime" ":"
           txEvts "," "waitingRequests" ":" waitQueue "]";

txEvts = "[ ]" | "[" txEvt {"," txEvt} "]";

txEvt = txRef ":" time;

waitQueue = "⟨ ⟩" | "⟨" request {"," request} "⟩";

xTxTimeoutC = "[ ]" | "[" txToEntry {"," txToEntry} "]";

txToEntry = "[" "tx" ":" txRef ["," "expireTime" ":" time] "]";
```

Transaction meta containers:

```
xTransactions = "[ ]" | "[" txProp {"," txProp} "]";

txProp = txRef ":" "[" "status" ":" string "," "log" ":" txLog "]";

txLog = "[ ]" | "[" opProp {"," opProp} "]";

opProp = opRef ":" "[" "capiName" ":" string "," "status" ":" string ","
                       "operationLog" ":" opLog "]";

opRef = label;

opLog = "⟨ ⟩" | "⟨" logProp {"," logProp} "⟩";

logProp = label ":" "[" "type" ":" string "," "path" ":" path "]";

path = label {"/" label};

xLocks = "[ ]" | "[" lockProp {"," lockProp} "]";

lockProp = label ":" "[" "type" ":" string "," "path" ":" path ","
                         "tx" ":" txRef ["," "op" ":" opRef] "]";
```