An Extensible Space-Based Coordination Approach for Modeling Complex Patterns in Large Systems[⋆,⋆⋆]

Eva Kühn, Richard Mordinyi, and Christian Schreiber

Vienna University of Technology, Institute of Computer Languages,
Space Based Computing Group and Complex Systems Design and Engineering Lab,
Argentinierstr. 8, A-1040 Wien
{eva,richard,cs}@complang.tuwien.ac.at

Abstract. Coordination is frequently associated with shared data spaces employing Linda coordination. But in practice, communication between parallel and distributed processes is carried out with message exchange patterns. What, actually, do shared data spaces contribute beyond these? In this paper we present a formal representation for a definition of shared spaces by introducing an "extensible tuple model", based on existing research on Linda coordination, some Linda extensions, and virtual shared memory. The main enhancements of the extensible tuple model comprise: means for structuring of spaces, Internet- compatible addressing of resources, more powerful coordination capabilities, a clear separation of user data and coordination information, support of symmetric peer application architectures, and extensibility through programmable aspects. The advantages of the extensible tuple model (XTM) are that it allows for a specification of complex coordination patterns.

1 Introduction

The coordination theory was founded by Malone and Crowston and described as "managing dependencies between activities". In [16] it is argued that coordination makes sense only if tasks are interdependent. Additionally, the theory suggests that standardized coordination mechanisms can be applied to specific coordination problems. Ciancarini therefore describes a generic coordination model [4] as a triple of {`E`, `M`, `L`}. It suggests to have a clear separation between the specification of the communication entities of a system and the specification of their interactions or dependencies. In the model, {`E`} stands for either physical or logical entities to be coordinated. These can be data (structures), software processes, services, agents, or even human beings interacting

[⋆] We would like to thank Stefan Craß, Geri Joskowicz, Hans Moritsch, Gernot Salzer, Thomas Scheller, Vesna Sesum-Cavic, and Ralf Westphal for their helpful discussions on this topic.

[⋆⋆] The project is partly funded by TripCom (IST-4-027324-STP project, http://www.tripcom.org) and CAPI (project at TU-Vienna) of the Institute of Computer Languages.

with computer-based systems. `{M}` represents the coordination media (i.e. communication channels) serving as a connector between the entities and enables communication, which is a mandatory prerequisite for direct coordination [26,6]. Such coordination media may be message-passing systems, pipes, tuple spaces etc. `{L}` specifies the coordination laws between the entities defining how the interdependences have to be resolved and therefore, semantically define the coordination mechanisms.

From the point of view of designing a language for distributed systems the idea of associative communication based on a shared data space is one of the most interesting paradigms [3]. This is because a shared space allows to clearly separate the issue of controlling coordinating communication entities from the issue to control a single entity. A Tuple Space [22] is an example of this kind of languages. It is a well-known coordination model such as Linda [8], JavaSpaces [7] and TSpaces [15]. Tuple spaces are flat and unstructured multisets of tuples that can be accessed via very basic output, read, and input operations.

In the tuple space approach processes communicate with the other entities in the environment by writing tuples (ordered sequences of data) into the tuple space. Sharing of data via spaces [2] is not a novel paradigm. It comes from parallel processing and was later considered for distributed environments. Due to its high-level abstraction of communication by simply reading and writing data from/into a shared space this paradigm fits to growing dynamics and collaboration in the network [29]. The processes interested in retrieving information useful for coordinating their activities perform blocking **rd** or **in** operations specified via a template. In case several tuples match the template of a data-retrieval operation, only one of them is selected non-deterministically.

The limitation of current tuple space implementations is that they support template-matching only. This is problematic if any other form of coordination is needed, like FIFO. In such cases the coordination entity itself has to manage ordering of the tuples the right way and agree about it with other entities. Thus, the implementation of the coordination entity must contain functionalities that a coordination media should provide. However, queries involving relational comparison operators cannot be implemented with template matching. The proposed approach aims to describe a generic and extensible coordination model based on tags, upon which any kind of coordination laws can be modeled, starting with simple ones like FIFO and KEY, to more complex concurrent collection patterns, and finally also patterns that as e.g. described in [9] also cope with distributon. The extensible coordination approach is realized by means of the coordination media XVSM (extensible virtual shared memory) [25], [14] clearly separating the responsibilities between coordination middleware and entity again. The reason for adding additional forms of coordination laws to a space coordination medium are based on our experiences with programming real applications:

– Developers consider Linda template matching as too unstructured in comparison to query facilities offered by databases. Being forced to pack coordination information into tuple content is a drawback [17].

– Autonomous business partners require a programming language and platform neutral API specification that allows coordination across the Internet. E.g., JavaSpaces [7] exhibits only a Java based API specification.
– For future Web2.0 scenarios, a crucial requirement will be to hide the differences between communication within an enterprise and across enterprise borders.

Experiences gathered with such applications within large systems are e.g. GONG with near-time database replication [12] using the Corso [11] virtual shared memory for load-balancing and network bandwidth optimization, the SVSDM [19], [18] used for the efficient distribution of working packages to mobile agents of an insurance company and in catastrophic scenarios [14], the RealSafe project for reliable data distribution and retrievement in the traffic telematics area, and the project SWIS [24] in cooperation with Frequentis AG[1] focusing on efficient and reliable clustering of network nodes and communication between those. These systems contain complex patterns like replication, notification, or fifo coordination. Therefore, this paper details in section 4 the way how such patterns are implemented with the introduced model.

In the following, the classic "tuple spaces model" referring to Linda and JavaSpaces is termed "TSM", and the "extensible tuple model" as proposed in this paper is called "XTM".

2 Structuring the Coordination Space

Motivated by numerous works that extend the original TSM, we propose an extension of the TSM called XTM (extensible tuple model) for modeling data and coordination in the shared space in a more structured way, and a formal notation for the XTM. The goal is to define a system with a minimal feature set that is extensible, can be configured upon request to meet complex collaboration requirements, and has a well-defined semantics. Briefly, the main concept in XTM is a shared container that manages data items called entries (generalizing tuples in TSM). Both, container and entry are modeled by means of an xtuple (extensible tuple). The operations on a container comprise addition, and (destructive) selection of entries. In this paper we show the query language of the XTM; the operational semantics are only sketched informally.

2.1 Xtuple and Entry

An xtuple (extensible tuple, or XT for short) is either an n-tuple "$\langle E1, \ldots, En \rangle$" which is an ordered list of entries Ei (i = 1..n) with the property that its arity n can change (expand or shrink) through addition and destructive selection of entries, or empty ("$\langle\rangle$"). Entries within an xtuple can be accessed using "XT[j]" where j is the position of the entry. An entry is a multi-set of tags. A tag is a pair that consists of a name and a value. The value of a tag can be accessed using

[1] http://www.frequentis.com

"get(name, E)" where name is the name of the tag and E the entry. Each entry in an xtuple always has one implicit tag representing a unique position number ("\$P") for access. The numbering within an xtuple starts with 1. However, the position number is not an immutable part of the entry, it may change if entries are added to or removed from the xtuple, and it is automatically stripped from the entry, if the entry is removed from the xtuple. Entries have no identity. The fact that entry E has a position number with value "i" is denoted as: "\$P=i".

Example 1. (xtuple)

- "⟨[payload=hello, \$P=1], [payload=space, \$P=2], [payload=2008, \$P=3]⟩" is an xtuple with 3 entries containing the payloads "hello", "space", and "2008" plus their position number tags.
- "⟨⟩" is an empty xtuple.
- Let E be [payload=eva], then "get(payload, E)" returns the value of the tag payload.

2.2 Container and Container Referencing

A container is a pair consisting of (1) a container-name which is a URL [14] that can be addressed via the Internet, and (2) an xtuple. For a container that is represented by the pair (C, xtuple), its xtuple can be accessed via "∗C".

A container-name is used by **read**, **take** and **write** operations to retrieve the container's xtuple at the peer site where the container is hosted. A distributed data space is a collection of such referenceable containers.

2.3 Structuring of Containers

Entries constitute either proxies for structured user data to be processed jointly, or coordination data for the interaction of processes.

2.4 Entry Addition

When an entry is added to an xtuple XT, it is appended to the end of the XT and the position number reflects the order in which the entries have been added.

Example 2. (entry addition)

- E1 = "[payload=jens, pin=12, title=prof]".
  XT1 = "⟨[payload=chris, \$P=1, pin=78, title=ms]⟩".
  Adding E1 to XT1 obviously yields "⟨[payload=chris, \$P=1, pin=78, title=ms], [payload=jens, \$P=2, pin=12, title=prof]⟩".

2.5 (Destructive) Entry Selection

For (destructive) selection of entries from an xtuple XT using **read** or **take**, an XQ (xtuple query) is introduced. Selected entries are copied into a new xtuple that represents the result of the (destructive) selection; take is destructive and also removes the selected entries from XT.

An XQ is a single-XQ (SXQ) or a composition of SXQs using the binary operator "|" which is left-associative and not commutative. A SXQ has the syntax, where "name" refers to the name of a tag:

- name *relational-operator* value (relational comparison)
- name $\in [value^1\text{–}value^k]$ (range)
- name $\in [value^1, \ldots, value^k]$ (enumeration)
- name $function$ (name, *template*) (matching function)

Beyond tag selection using relational comparison operator, range or enumeration definitions, also a *match-maker* function can be given, which denotes a user-definable function. Its argument represents a template that is matched with the value of the name and results into success or fail.

The execution semantics is that each SXQ (from left to right) of an XQ = SXQ_1 | ... | SXQ_n is applied to each entry in the xtuple XT in entry position order. If successful, a new xtuple results to which the next SXQ is applied, a.s.o. The basic idea behind this mechanism is that streaming becomes possible, as each SXQ can be decided by each single entry. In contrast to databases, no join and aggregation operators are supported. The SXQs can work concurrently and can be seen as filters that pump the information from one stage to the other, comparable with the staged driven architecture (SEDA) approach [28], [27] that is known to scale well. Figure 1 depicts the execution of an XQ. The first SXQ, SXQ_1, is applied to the input xtuple XT creating an xtuple (XT') that contains all entries which fulfil the SXQ. XT' is then the input for the next SXQ, SXQ_2. This process is repeated for all SXQs of the XQ. After the last SXQ_n has been executed the resulting xtuple (RXT) contains all entries which fulfil the XQ.

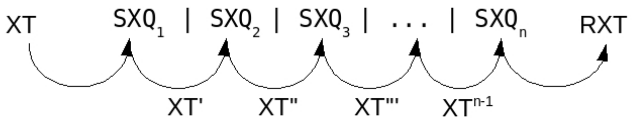

Fig. 1. Execution semantic of an XQ

An entry fulfills an XQ, if it fulfills all SXQs of the XQ. An entry fulfills a SXQ:

- If the SXQ is a relational comparison, or a range or enumeration query and if the entry has at least one tag with the given name the value of which fulfills the SXQ. For a relational comparison, or a range query the result could be 0,1,... entries, whereas for an enumeration query at least one entry must be found for each value contained in the enumeration. Note that $ is used to denote the arity of the xtuple and can be used in query expressions.
- If the SXQ is a match-maker function and a user-defined match-maker function exists with same name as the match-maker function name, which applied to the denoted tag's value of the entry returns success. This means that the entry must possess at least such a tag. The parameters of the user defined match-maker function are the name of the value to which it is to be applied, and the template.

The result of an application of a SXQ contains all entries that fulfill the XQ and that are currently contained in the xtuple to which it is applied. An XQ is used by **read** and **take** for selection and is executed in one atomic step. If later on entries are added to the xtuple that would also fulfill the XQ, they are not considered any more.

Example 3. (fullfillment).

– Entry "[payload=A, a=5]" fulfills the XQ "a∈[1 – 100]".
– Entry "[payload=B, b=5]" fulfills the XQs "b∈[3, 5]", and "b=∗". Note: with "=" the usage of ∗ as wildcard is allowed.
– Entry "[payload=C, c=1, d=2]" fulfills the XQ "d∈[2 – 4] | c=1".

If a SXQ cannot be fulfilled, the **read** or **take** is delayed (except if there are contra-dictionary SXQs in which case it fails), and next time will retry from the beginning, i.e. with the left-most SXQ. E.g. selection from XT = "⟨[payload=100, link=left], [payload=200, link=right]⟩" using XQ = "link∈[next]" will delay.

Example 4. (SXQ, XQ)
Selections that could cause a delay are marked with "D"; "n" is the arity of the XT to which the XQ is applied; column "nres" denotes number of result entries.

XQ:	select from XT:	nres:	delay:
$P ∈ [1,...,10]	the first 10 entries	10	D
pin ∈ [1 – 10]	all entries with a tag named "pin" with value between 1 and 10	$0, 1, \dots, n$	
pin ∈ [1,. . . ,10]	all entries with a tag named "pin" with value between 1 and 10; at least one entry must be found for each given value	$10, 11, \dots, n$	D
payload ∈ linda((chris,*,*)) | pin ≤ 50	all entries that represent students with name "chris" and that have a pin that is ≤ 50	$0, 1, \dots, n$	
P ∈ [1 – 10] | label ∈ [4 – 5]	the up to 10 entries in XT, from these select all that have a label with value "4" or "5"	$0, 1, \dots, 10$	
l1 = green | l2 = blue | payload ∈ (ralf,author,∗) | x ∈ [5,6] | $P ∈ [1,2]	all entries that have a tag with name l1 and value="green", and that have a tag with name l2 and value="blue", and that have a palyoad tag that is a tuple with arity="3" with first arg="ralf" and second arg="author", and that have a tag with name x and value="5" and one with value="6" (note: entries for both keys must be found), and select two entries in position order	2	D
P ∈ [1 – $]	all, possibly 0 entries	0,1,. . .,n	

3 Core Container Functionality

A container exhibits a lean API which is defined by means of an XML protocol [5]. No connect operation is required as each operation refers to an Internet-addressable resource. Each peer application runs an embedded space runtime. Each protocol message also carries a context parameter not shown below, in which system or user defined parameters can be passed: e.g. security tokens, execution states etc. In the following, the protocol messages[2] defining the basic core container functionality, and their operational semantics are only informally described:

***i*-tx-create (S, ttl):** creates a local transaction at the peer's site denoted by the URL "S" which refers to the URL of a peer runtime representing a local space; "*i*" stands for isolation-level and is either "e" or "c": An exclusive transaction ("e") locks every container it accesses exclusively, whereas a concurrent transaction ("c") allows concurrent readers and writers using pessimistic locking at entry level. The "ttl" parameter sets a time-to-live value for the entire transaction in seconds; if expired the transaction is automatically rolledback. The answer protocol message contains the newly created transaction reference ("tx") which is a URL, too.

tx-commit (tx), tx-rollback (tx): commits or rollbacks a transaction given by its URL "tx". The answer is the termination result of the transaction.

create-container (S, C, tx): creates a container with the name "C" in a space referenced by its URL "S", within a transaction.

destroy-container (C, tx): destroys a container with a given name "C" within a transaction.

***read-op* (C, XQ, tx, to, AC):** (destructively) selects entries from a container (with URL "C") with optional transaction "tx" and timeout "to" parameters; "*read-op*" stands for either take or read. If tx is null, an implicit tx is assumed that immediately commits. A timeout > 0 indicates that this operation might block and retry within the given timeout if no answer is found. Answer to a **take** or **read** message is a **write** message (see below) that writes the selected entries into container "AC". Typically AC will be located at the requestor's site and can e.g. be intercepted by a pre-write aspect (see below) that calls a callback method or wakes up some waiting thread. "AC" can be null, in this case the result is not written anywhere. A take operation with "AC" set to null corresponds to a delete operation and can be used to save network bandwidth. Note that we describe a protocol; a binding can be provided for any programming language which can use this protocol to implement a blocking read/take or an asynchronous callback.

write (C, E, tx, to): adds entry E to container C.

bulk-write (C, <E1, E2, ..., Ek>, tx, to): appends k entries E1, E2, ..., Ek—given by means of an xtuple—to container C in the order of the xtuple.

[2] The protocol is implemented in XML which allows the XTM to be used by any programming language binding.

A main concept of the XTM are *aspects* to extend the space dynamically through user defined actions, i.e. a script or method call that are injected into any of the above operations at a certain ipoint (interception point). This is comparable to reactions in LIME [20], [21]; the difference is that aspects are called before (pre) a certain operation is invoked on a container, immediately afterwards (post), or if the operation is committed (on-commit); whereas reactions are called when a tuple matching a certain pattern is found in a tuple space.

def-aspect (C, ipoint, priority, program, tx, ttl): creates an aspect and in the answer message sends a URL (aspect identifier "aid") to identify it. Program is an xtuple containing the single lines of the script to be executed. The ttl parameter specifies its life time.
undef-aspect (aid, tx, to): removes an aspect given by its URL.

The order of aspect action execution is resolved by a priority parameter. Cf. join points in aspect oriented programming which allow triggering of code before, after, or instead of method execution [10]. This concept has been proven useful for collaborative space-based applications at application level [13] and is now proposed to be integrated into the space core.

With aspects, the behaviour of a container can be changed creating a shareable "abstract data type" based on a container. Orca [1] showed that "having users define their own operations has many advantages". Orca supports shared, replicated objects that can be read and updated using blocking and non-blocking read/write operations. However, in Orca no dynamic "programming" of the space is possible.

Aspects can also be used to implement reactions on state changes [20], [21]: e.g. if a certain entry is written to, or taken from a container.

An action has access to parameters and relevant data of the original method it intercepts and depends on its ipoint. E.g. a pre-write action has access to 5 parameters referencable by $\$0, \$1, \ldots, \$4$: with \$0=context, \$1=C, \$2=entry, \$3=tx, and \$4=to; a post-write-on-commit action or a post-write action has: \$0=context, \$1=C, \$2=entry, \$3=tx, and \$4=to. For the bulk-write ipoints, \$2=xtuple with the written entries. Read ipoints have the following arguments: \$1=C, \$2=XQ, \$3=tx, \$4=to and \$5=CA.

The return value of an aspect determines the control flow: "ok" implies normal continuation; "skip" in a pre-ipoint skips all further pre-actions and the original method action, continuing with the first post-action, if given; "skip" in a post-action skips all further post-actions; "fail" causes the entire operation to fail; "reschedule" causes the operation to be delayed until a corresponding event occurs upon which it is rescheduled, as long as the timeout condition is fulfilled; and "remove-me" causes the aspect to be removed. Execution states are implicitly passed via context parameter.

The main goal was to keep the set of operations of the XTM minimal and extensible so that higher-level APIs for more advanced data structures and collections can be bootstrapped on top of it. For the extensibility aspects are introduced. The core container functionality like read, write and take is influenced by the Linda model. All of these support bulk-data to optimize the network behaviour and to enable the implementation of iterators. Basic transaction functionality is obviously essential and transactions have therefore already been suggested by other spaces like JavaSpaces (pessimistic concurrency control) and Corso (optimistic concurrency control). The XTM provides pessimistic concurrency control.

4 Extending the Functionality

With help of the XTM functionality, more complex coordination laws can be defined and "injected" into the space without changing the protocol. This can be for example FIFO selection, or more complex access patterns like a time bounded cache, notifications and iterators, or other vertical features like replication. Also more complex coordination laws can be defined this way, like e.g. a container supporting multiple coordinations simultaneously.

In the following code snippets, "result $\leftarrow$ message" stands for: "send message, execute it, and extract result from the answer message of the protocol".

4.1 FIFO Coordination

Example 5. (FIFO coordination) Define a new SXQ named **FIFO(n)** that selects n entries in FIFO order and that can be used by **read** or **take** operations:

FIFO(n) ::= \$P∈[1,...,n]

4.2 Notification

In this section two realisations of a notification mechanism are shown. At least two containers are normally involved in a notification in XTM. The container on which the notification is registered and a container which stores the information of the fired notifications (the notification container). A notification is realized by analysing the operations executed on a container using an aspect. The aspect decides whether the notification has to be fired or not. For example, a notification could fire when a template matches newly written entries, a certain amount of entries has been written, or when entries have been read. In case of the notification has to be fired, the aspect writes the required information into the notification container. From this container a user can take the information for further computation (waking up a thread, calling a callback method, ...). The information written to the notification container can be customized for the needs of an application. For instance, the entry which caused the notification to fire, a token which indicates that the notification fired or any other information can be written by the aspect.

Example 6 defines a notification similar to the JavaSpace [7] notification. It fires when a newly written entry matches a given template. The new entry which caused the notification to fire is not passed to the application. The application is only informed that something new has been written.

First, a container (TC) is created which stores the template and an aspect is created which handles the firing of the notification. The aspect simply executes a read operation on the TC after each (committed) write operation using the payload of the new entry as template. This is possible because it is assumed that the used match-maker function (LINDA) can handle the inversion of the template and the payload. The answer of the read operation is written into the notification container (in this case the answer of the read operation is always the template because it is the only entry in the TC).

This example could be extended supporting multiple templates by writing more then one template into the TC. The user can distinguish which template caused the notification to fire by analysing the data in the NC.

Example 6. (JavaSpace notification)

```
create-Notification(site, C, CN, template) :=
    create-container(site, TC, null)
    write(TC,⟨[payload = template]⟩, null, 0)
    def-aspect(C, post-write-on-commit, priority,
        ⟨ read(TC, LINDA(payload, get(payload, $2), null, 0, NC) ⟩,
        null, INFINITE)
```

Example 7 defines another notification flavour which is very similar to the first one. Instead of informing the application that something happened, the entry which caused the notification to fire is written to the notification container. To realize this an aspect is added to the container on which the notification shall be registered. When a new entry is written, the aspect creates a new container containing the new entry. Afterwards, a take operation is executed on the new container using the template passed during creation of the notification. The result of the take operation is written to the NC (when the take operation can be fulfilled the entry matches the template and therefore the notification has to fire). Finally, the container created by the aspect is destroyed to avoid unnecessary garbage.

Example 7. (Notification which returns the data)

```
create-Notification(site, C, CN, template) :=
    def-aspect(C, post-write-on-commit, priority,
        ⟨ create-container (site, C', null)
        write(C', ⟨$2⟩, null, 0)
        take(C', LINDA(payload, template), null, 0, NC)
        destroy-container(C') ⟩,
        null, INFINITE)
```

4.3 Read Iterator

Example 8. (iterator) Define an iterator that (1) first reads all existing entries from a container and then (2) asynchronously copies all prospectively inserted entries into a container "CN" from which these entries can be collected with

take. (2) is achieved by means of an aspect. The aspect writes all entries written to C into the answer container CN.

The reading of all data (1) and the aspect definition (2) must be done in one exclusive tx to avoid data loss.

```
read-iterator (site, C, CI) ::=
    tx ← e-tx-create (site, INFINITE)
    read (C, FIFO($), tx, CI)
    def-aspect (C, post-write-on-commit, priority,
        ⟨ write (CI, $2, $3, $4) ⟩,
        tx, to, INFINITE);
    tx-commit (tx);
```

To keep the above examples simple, bulk-operations are not considered. The required extensions are straight forward and can be done analogously to the specification of the replication shown in the next section.

4.4 Single Master Replication

Finally we show a pattern for a single master replication. One peer site is the owner of the master copy of the container (MC), and other sites can create replica containers RC_1,...,RC_k from M. From each RC_i in turn new replicas RC_{i1},...,RC_{ik} can be created a.s.o. as depicted in figure 2.

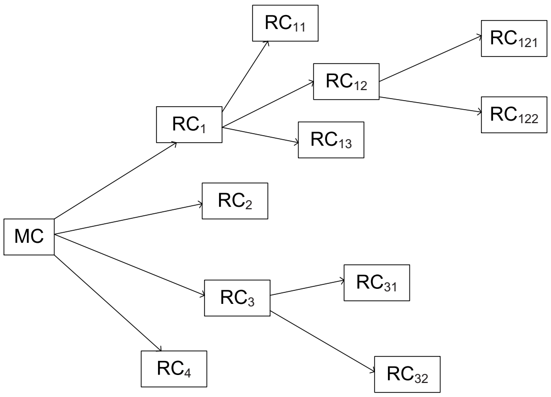

Fig. 2. Single Master Replication

Each RC shall be an identical copy of the MC. It represents a cache from which peers can read in order to reach a scalable system, and to improve availability and performance. If a peer wants to write to a local RC, then there are two scenarios in the pattern: (a) this is forbidden, and (b) the writing is allowed but internally redirected to the container from which the RC was created. If RC is not a direct ancestor of MC, then the re-direction is done in a cascading way. In the following examples we show the re-direction as a “write-through” operation that is perfomed synchronoulsy. A variant that calls the aspect in a fire-and-forget way would allow for an asynchronous behaviour.

The newly defined extended API is termed **create-single-master-replica-container (MC, RC)**. The name RC must not yet be used. A new container is created and published under the name RC, all entries contained in MC are copied into RC (called *first sync*) and an aspect is injected into MC that listens on changes and transfers and applies them to RC. For this, an on-change-iterator is defined that performs the following actions in an atomic step (see figure 9):

1. Create aspects on MC for the post-ipoints of all operations that perform changes on a container—i.e. write, bulk-write, and take—to report the information about the change into a container called RSC (see below) that is passed to the API as argument and that serves to collect the changes for each peer replica, in the form of a *command entry.* The command entry has the form [op=*command*, changed=*entries*], and otherwise [op=*command*, inserted=*entries*], where *command* is either delete or insert. For the replication scenario, it is assumed that each entry that is inserted into MC possesses a unique id with tag-name “guid”. So in order to be a capable MC, it must possess an aspect that is injected upon container creation and that adds this tag to each newly written entry.
2. Copy all entries E1,. . .,Ek of MC into RSC as a command entry of the form [op=bulk-write, inserted=⟨E1,. . .,Ek⟩].

Example 9. **on-change-iterator (MC, RSC, tx) ::=**

```
read (MC, FIFO($), tx, INFINITE, RSC)
def-aspect (MC, post-take, 1, ⟨ write (RSC, [cmd=delete, changed= ∗$5],
    $3, $4) ⟩, tx, INFINITE, INFINITE)
def-aspect (MC, post-write, 1, ⟨ write (RSC, [cmd=insert, changed= $2],
    $3, $4) ⟩, tx, INFINITE, INFINITE)
def-aspect (MC, post-bulk-write, 1, ⟨ write (RSC, [cmd=delete, changed=
    ∗$2], $3, $4) ⟩, tx, INFINITE, INFINITE)
```

We can now implement the replication to perform (see figure 10) the following steps atomically, where we assume that the RSC shall be created at the site of the MC:

1. Create a new container termed *replication subscription container* RSC for RC. RSC can be located at either the site of MC, or RC, or at any third site. In the first case, this refers to a “pull” situation, and in the second case it implements a “push” scenario.
2. Create an on-change-iterator for this new replica on MC.
3. Periodically fetch the changes in SRC and apply them to RC. Note that this could also lead to variants of the pattern where the fetching is done e.g. at certain times, at certain events (using an aspect), or taking only a certain amount of changes in one step etc.

Example 10. **create-single-master-replica-container (MC, RC) ::=**

```
tx ← tx-create (MC-site, INFINITE)
create-container (MC-site, RSC, tx)
on-change-iterator (MC, RSC, tx)
tx-commit (tx)
periodically do:
    create-container (RC-site, TEMPC)
    take (RSC, $P = 1 , tx, INFINITE, TEMPC)
    if(get(cmd, *TEMPC[1]) == delete)
        take(RC, guid=entry.guid, null, 0, null)
    else
        write(RC, get(changed, *TEMPC[1]), null, 0)
    destroy-container (TEMPC)
```

For case (a) where updates are forbidden on a RC, upon creation of the RC pre-aspects must be set on every change operation that simply return an exception without calling the original operation. For case (b), these aspects are implemented differently in that they re-direct the operation as is to the direct ancestor container from which the replica RC was created.

5 Conclusion

Reliable, near-time, on-line collaboration will require a different architectural style than a “store & forward” oriented one. Bi-directional access to arbitrary data structures that go beyond fifo queues are needed to ease collaboration.

In this paper we have presented a formal notation called the extensible tuple model (XTM) that can serve to model the TSM (tuple space model) as well as essential enhancements of it. XTM allows for a clear specification of complex coordination patterns. Examples are shown for FIFO coordination, two different notification patterns, an iterator pattern, and a replication pattern with some variants. The main proposed extensions are a better structuring of the space, more powerful coordination capabilities, a clear separation of user data and coordination information, support of symmetric peer application architectures, a well- defined semantics of bulk-operations (read/take of multiple entries), and a protocol that makes it usable in the Internet. The model has been implemented by the XVSM system and is available as an open source implementation [25].

XTM is extensible with respect to the following three points: (1) The behavior of a container, which is the core shared data structure in the XTM, can be programmed dynamically using aspects. This way, arbitrary abstract data types can be created which resemble shared collections for queues, dictionaries, maps, trees etc. Using aspects, notifications and iterators in different flavors can be realized with a well-defined semantics. (2) The possibility to include coordination tags into an entry can be used to extend the container by other higher-level user defined selectors, e.g. stack, random access, vector, least recently used, priorities, role based scheduling etc. (3) The XTM query language can be extended by the definition of match-maker functions, e.g. for supporting RDF or XML based query facilities.

The described model in this paper has been used to implement the open source XVSM implementation called MozartSpaces [23]. In our future work we will use this model for verification and analyzation of space implementations as described in section 4. Additional future work will also deal with further extensions of the XTM query language, and a model for the operational semantics of the XTM.

References

1. Bal, H.E., Kaashoek, M.F., Tanenbaum, A.S.: Orca: a language for parallel programming of distributed systems. IEEE Transactions on Software Engineering 18(3), 190–205 (1992)
2. Carriero, N., Gelernter, D.: Linda in context. Commun. ACM 32(4), 444–458 (1989)
3. Ciancarini, P.: Distributed programming with logic tuple spaces. New Gen. Comput. 12(3), 251–284 (1994)
4. Ciancarini, P.: Coordination models and languages as software integrators. ACM Comput. Surv. 28(2), 300–302 (1996)
5. Ecker, S.: Communication protocols in XVSM - design and implementation. Master's thesis, Vienna University of Technology, E185/1 (2005)
6. Franklin, S.: Coordination without communication. Technical report, Inst. For Intelligent Systems, Univ. of Memphis (April 2008)
7. Freeman, E., Arnold, K., Hupfer, S.: JavaSpaces Principles, Patterns, and Practice. Addison-Wesley Longman Ltd., Essex (1999)
8. Gelernter, D.: Generative communication in linda. ACM Trans. Program. Lang. Syst. 7(1), 80–112 (1985)
9. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
10. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
11. Kühn, E.: Virtual Shared Memory for Distributed Architecture. Nova Science Publishers (2001)
12. Kühn, E.: The zero-delay data warehouse: mobilizing heterogeneous database. In: Proceedings of the 29th international conference on Very large data bases (VLDB 2003), pp. 1035–1040 (2003)
13. Kühn, E., Fessl, G., Schmied, F.: Aspect-oriented programming with runtime-generated subclass proxies and net dynamic methods. Journal of NET Technologies 4, 1801–2108 (2006)
14. Kühn, E., Riemer, J., Mordinyi, R., Lechner, L.: 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)
15. Lehman, T.J., McLaughry, S.W., Wycko, P.: T-spaces: The next wave. In: HICSS (1999)
16. Malone, T.W., Crowston, K.: The interdisciplinary study of coordination. ACM Comput. Surv. 26(1), 87–119 (1994)
17. Martin, D., Wutke, D., Scheibler, T., Leymann, F.: An eai pattern-based comparison of spaces and messaging. In: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), Washington, DC, USA, p. 511. IEEE Computer Society Press, Los Alamitos (2007)

18. Mor, M., Mordinyi, R., Riemer, J.: Using space-based computing for more efficient group coordination and monitoring in an event-based work management system. In: The Second International Conference on Availability, Reliability and Security (ARES 2007), pp. 1116–1123 (April 2007)
19. Mordinyi, R.: Shared virtual space distribution manager - SVSDM - design and implementation. Master's thesis, Vienna University of Technology, E185/1 (2005)
20. Murphy, A.L., Picco, G.P., Roman, G.-C.: Lime: A coordination model and middleware supporting mobility of hosts and agents. ACM Trans. Softw. Eng. Methodol. 15(3), 279–328 (2006)
21. Picco, G.P., Murphy, A.L., Roman, G.-C.: Lime: Linda meets mobility. In: ICSE 1999: Proceedings of the 21st international conference on Software engineering, pp. 368–377. IEEE Computer Society Press, Los Alamitos (1999)
22. Semini, L., Montangero, C.: A refinement calculus for tuple spaces. Science of Computer Programming 34(2), 79–140 (1999)
23. MozartSpaces WebSite (2008), `http://www.mozartspaces.org`
24. SWIS WebSite (2008), `http://www.isis.tuwien.ac.at/node/4841`
25. XVSM WebSite (2008), `http://www.xvsm.org`
26. Weigand, H., van der Poll, F., de Moor, A.: Coordination through communication. In: Proc. of the 8th International Working Conference on the Language-Action Perspective on Communication Modelling (LAP 2003), pp. 1–2 (2003)
27. Welsh, M., Culler, D.: Overload management as a fundamental service design primitive. In: EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop, pp. 63–69. ACM Press, New York (2002)
28. Welsh, M., Culler, D., Brewer, E.: Seda: an architecture for well-conditioned, scalable internet services. SIGOPS Oper. Syst. Rev. 35(5), 230–243 (2001)
29. Zhen, L., Parashar, M.: Comet: a scalable coordination space for decentralized distributed environments. In: Second International Workshop on Hot Topics in Peer-to-Peer Systems (HOT-P2P 2005), 21 July 2005, pp. 104–111 (2005)