OPENSPACES

An Object-Oriented Framework for Configurable Coordination of Heterogeneous Agents

Diploma Thesis

of the Faculty of Sciences
University of Berne

by

Thomas F. Hofmann

April, 2001

Supervisor:

Prof. Dr. Oscar Nierstrasz

Institute of Computer Science

Typesetting made with LaTeX & *pleasure* :-)

Author's address:

Software Composition Group
University of Berne
Institute of Computer Science and Applied Mathematics (IAM)
Neubrückstrasse 10
CH-3012 Bern
Switzerland

mailto:hofmann@iam.unibe.ch
http://www.iam.unibe.ch/~hofmann/

Abstract

Tuple spaces have turned out to be one of the most fundamental abstractions for coordinating software agents. They offer a simple and natural way of communication and are capable to express a large class of distributed and parallel algorithms.

While many extensions to the original Linda model have been proposed, no one approach seems to be universally applicable to all problem domains. In this thesis we investigated how a tuple space can be extended to support *configurability of its behavior*. In this way, several variants of the coordination model can be realized without changing the underlying base system. Moreover, charging tasks to the coordination medium allows a programmer to implement an application at any desired level of abstraction.

A prototype framework, OPENSPACES, has been developed with the object-oriented language Smalltalk. It supports both static configurability as well as dynamic reconfiguration of the behavior policies through runtime composition.

To be useful in open distributed systems, a coordination medium must be capable of coordinating a variety of different software entities. OPENSPACES therefore is built on top of CORBA and provides access for *heterogeneous external clients*. It can be used from any platform using any programming language with a CORBA implementation. The sole prerequisite for participating in a OPENSPACES-based application is the implementation of the small IDL interface. Hence not only the provided standard clients, but any external software agent may be coordinated.

We present the framework and show with a set of typical examples how it can be instantiated and configured for different and changing needs. As an example of a heterogeneous setup with external clients, a Java agent has been developed to participate in one of the example applications.

Acknowledgements

There are a lot of people who helped me in the course of writing this thesis. First of all, I'd like to thank my supervisors for their support during this project: *Stéphane Ducasse* for pushing me with great enthusiasm and profound Smalltalk knowledge, due to which I've even seen Cyprus and Southampton :-) *Juan Carlos Cruz* for being the coordination and tuple space guru and inspiring me to many experiments with distributed hacking. Professor *Oscar Nierstasz* for being the head of the coolest group of the CS department, and for his constructive questions and comments on my work, and also for his participation in writing the paper for Coordination 2000.

Working in the SCG was fun! My thanks go to *all* members, especially to: *Matthias* for many discussions on games and more, and for his sound review of the predecessor of this writing, costing him one red and half a green pencil... *Franz* for his patience when explaining Piccola and LaTeX to me, and his superior coordination of Coordination. *Georges* for ,,Gipfeli" support, and discussions about the meaning of life and work. *Lukas* for his valuable tips concerning optimization of the Vaio. *Jean-Guy* for his – *free* – CORBA tips and support.

Last but not least, special thanks to: My parents, *Eva-Maria* and *Felix*, for giving live to me and supporting me. *Erika* for enduring the ups and downs, and still being my sunshine. *All* my friends for playing music, discussing, flying kites, drinking beers with me, and for all the fish...

Thomas Hofmann
April, 2001

Contents

Abstract iii

Acknowledgements iv

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Problems . . . 2
- 1.3 Contribution of OPENSPACES . . . 3
  - 1.3.1 Configurability . . . 3
  - 1.3.2 Heterogeneity . . . 4
- 1.4 Structure of the thesis . . . 4

2 Coordination and LINDA 5
- 2.1 Coordination . . . 5
  - 2.1.1 Need for Coordination . . . 5
  - 2.1.2 Definitions . . . 6
  - 2.1.3 Coordination Models . . . 6
- 2.2 LINDA . . . 7
  - 2.2.1 Primitives . . . 7
  - 2.2.2 Matching . . . 8
  - 2.2.3 Active Tuples . . . 8
  - 2.2.4 Usage scenario . . . 9
- 2.3 LINDA as a Coordination Model . . . 11
  - 2.3.1 Advantages . . . 11
  - 2.3.2 Limitations . . . 12

3 Evolution of the Tuple Space Model 14
- 3.1 Overview . . . 14
- 3.2 Extensions to LINDA . . . 15
  - 3.2.1 Multiple Tuple Spaces and Distribution . . . 15
  - 3.2.2 PARADISE . . . 17

3.2.3 BONITA . . . 17
3.2.4 Multiple Read Problem . . . 18
3.3 Object-Orientation . . . 18
3.3.1 Matsuoka and Kawai . . . 18
3.3.2 OBJECTIVE LINDA . . . 19
3.3.3 BERLINDA . . . 19
3.3.4 JADA . . . 20
3.3.5 JAVASPACES . . . 20
3.3.6 TSPACES . . . 21
3.4 Programmability . . . 21
3.4.1 LAW-GOVERNED LINDA . . . 21
3.4.2 PROGRAMMABLE COORDINATION MEDIA . . . 23
3.4.3 LUCE . . . 23
3.4.4 Drawbacks of existing approaches . . . 24

4 The OPENSPACES Framework 26
4.1 Overview . . . 26
4.2 The Core Classes and their Relationships . . . 27
4.2.1 Entries . . . 28
4.2.2 Configuration Policies . . . 28
4.2.3 Spaces . . . 28
4.2.4 Space Agents . . . 30
4.2.5 The Space Server and Its Administrator . . . 30
4.3 The Access Contracts . . . 31
4.3.1 Intercepting the Access Operations . . . 31
4.3.2 The Read Contract . . . 31
4.3.3 The Take Contract . . . 33
4.3.4 The Write Contract . . . 33
4.3.5 Benefits . . . 33
4.3.6 Caveats. . . . 34
4.4 Implementation . . . 34
4.4.1 Registering Configuration Policies . . . 34
4.4.2 Retrieving Operations . . . 35
4.4.3 Blocking Operations . . . 36
4.4.4 The `write` Operation . . . 37

5 Case Studies 39
5.1 An Electronic Market Place . . . 39
5.1.1 Scenario . . . 39
5.1.2 Analysis . . . 40
5.2 Market Place V.1: Standard Implementation . . . 41

5.2.1 Using the Market Place forms . . . 42
5.2.2 Uniquely Identifiable Entries . . . 43
5.2.3 Stepping through a trade . . . 44
5.3 Market Place V.2: Consistency Assertions . . . 45
5.4 Market Place V.3: Automatic Index Handling . . . 47
5.5 Market Place V.4: Garbage Collection . . . 48
5.5.1 Reconfiguration . . . 50
5.6 Monitoring Space Accesses . . . 50
5.7 Protecting Single Entries . . . 52
5.8 Characteristics of the Examples . . . 53

6 Distribution and Heterogeneity 55
6.1 Distributed Applications . . . 55
6.2 Introduction to CORBA . . . 56
6.3 The CORBA–Layer of OPENSPACES . . . 57
6.3.1 Basics . . . 58
6.3.2 IDL Definitions for OpenSpaces . . . 58
6.3.3 Impact on the Design of Applications . . . 59
6.4 The Java Client . . . 61
6.4.1 IDL–to–Java . . . 62
6.4.2 ANY overhead . . . 64
6.4.3 Complications with Type Codes . . . 65
6.5 Distributed Event Service . . . 68
6.5.1 Notification of Space Clients . . . 69
6.5.2 Market Place V.5: Automatic Notification . . . 69

7 Conclusions 70
7.1 Need for Configurability and Heterogeneity . . . 70
7.2 OPENSPACES's Solutions . . . 71
7.2.1 White-Box Extensions . . . 71
7.2.2 Defining Configuration Policies . . . 72
7.2.3 Run-Time Configuration . . . 72
7.2.4 Heterogeneity . . . 73
7.3 Perspectives . . . 74

Chapter 1

Introduction

In this thesis we investigate how tuple spaces can be made *configurable* in their behavior and, at the same time, accessible for *heterogeneous* external clients.

1.1 Context

Modern software tends to be constructed as *composition* of several autonomous components (or *agents* [1]) running on the same computer or distributed over the network [Nie93]. This approach has many advantages like exploiting resources spread over the network, ease of adaptation to changing requirements, reuse of parts accomplishing typical tasks, and many more.

The composition paradigm leads to the need of *coordinating concurrent activities*. In order to cooperate the single components need to synchronize and exchange information which requires some suitable *communication structures*. Traditional paradigms for inter-process communication are: message passing, remote procedure calls (RPC) and distributed shared memory (DSM) (see e.g. [CDK94], [Kie97]). Message passing and RPC both have the disadvantage of *tight coupling* of the participants. The receiver of a message must be known to the sender, e.g. machine address, portnumber and process ID. The communication is inherently synchronous: the sender blocks until the call returns. Using DSM processes need not to know each other, however they still must know the location of the data in the memory structure. Moreover DSM abstractions produce a significant overhead by synchronizing the memory images of all participants and are therefore difficult to use.

A successful alternative is the concept of *tuple spaces*, first introduced by Gelernter and Carriero with the Linda system [GC85]. A tuple space is a special

[1] For the purpose of this thesis we use the general term *agent* for any software component or process participating in an ensemble forming a collaborative application.

form of a shared memory abstraction using but one central repository that can be accessed by all agents. It allows them to exchange tuples that contain any information they need to communicate. The sender inserts a tuple into the space, the receiver retrieves it associatively by providing a template tuple which describes the desired tuple. This *„generative communication"* style provides both *spatial and temporal decoupling* of the participants. The agents do not need to know their mutual location, the only requirement is the common knowledge of the communication space. Messages once put into the space may be retrieved at any time later; not even co-existence of the agents is needed.

The *decoupling* properties are highly attractive for open distributed systems where participating agents or components may be added dynamically. Compared to the traditional approaches the integration of components is more flexible, their setup and configurations are easier. The necessary access operations are simple and elegant, three basic primitives are sufficient to use the model: `write` to put a tuple to the space, `read` or `take` to copy or consume respectively a tuple matching a given template. Since the retrieving operations are associative, agents specify *what* they are looking for and not *where* to find it.

Coordination. LINDA founded a new research area in computer science: *Coordination* investigates how to organize concurrent activities that cooperate to form a new application or system. A *coordination model* helps to encapsulate details of the necessary communication and allows a developer to implement applications on a more abstract level. Several models have been introduced and are used successfully. In chapter 2 we give an overview of LINDA-like coordination models.

1.2 Problems

Since the proposal of LINDA, tuple spaces have been very successful and many variations and extensions to the standard model have been introduced (see chapter 3). It would be desirable, however, to have one system that could be *configured* to different needs, instead of changing the whole system for additional features.

While the simplicity of tuple space communication provides for high flexibility and universality, the resulting protocols for using them are often a bit awkward, potentially inefficient, and not secure. There is no way to enforce clients to only use the space according to the rules of an application. If an agent – maliciously or not – writes or consumes the wrong tuples, an application can become corrupt or may even crash. A typical example is a counter tuple that guarantees a consistent indexing of the tuples. If it is missing, the application cannot continue normally.

It would be useful to have some functionality to facilitate the protocols by *delegating more responsibility* to the coordination medium. If the tuple space itself can become active, it may validate accesses and keep its contents in a con-

sistent state. If the space can be charged with helper tasks like incrementing a counter tuple, the resulting architectures become more flexible and robust. Moreover, freeing the coordinated agents from such low-level tasks helps developers to implement their application over any desired level of *abstraction*, which again facilitates the development of participating agents.

There are few models offering a certain degree of configurability in the sense mentioned above. They are presented in section 3.4. Still they suffer from some limitations:

- *Internal Agents.* Only agents from within the systems may use the spaces. This is not enough in systems aiming to be open. A true generic communication medium must offer access for any kind of participant capable of performing the protocol.
- *Client Implementation Languages.* It should be possible to write agents in any language that suits their respective needs best. This also helps to reuse existing software components.
- *Programming the Medium.* The existing approaches are restricted to logic programming systems: reactions of a space may only be expressed with Prolog-like dialects. While this is sufficient in terms of (Turing) completeness, the practical integration of the models becomes highly complex.
- *No Object-Orientation.* For good reasons, object-orientation is today's preferred programming paradigm. It would be desirable to have an object-oriented configurable coordination model.

1.3 Contribution of OPENSPACES

For this thesis we developed OPENSPACES, an object-oriented framework that realizes the above mentioned requirements. It offers the core services of tuple spaces as standard features, and at the same time allows the behavior policies in place to be arbitrarily tailored. They can be activated both before starting the system, and also dynamically at run-time. OPENSPACES can be used for coordination in heterogeneous environments, it provides access to any external clients.

1.3.1 Configurability

OPENSPACES is both a *white-box* and a *black-box* framework [JF88]. The tuple space abstraction `OpenSpace` can be specialized to support new features like new access operations. The class `ConfigurationPolicy`, the encapsulation of an `OpenSpace`'s behavior, can be extended to add specific actions triggered by accessing the space. Its instances can be activated dynamically at run-time.

OpenSpaces can be tailored in the following ways:

- Different kinds of entities to be stored in an `OpenSpace` are defined by subclassing the generic tuple class `Entry`.
- To specify the lookup algorithm for the associative retrieval, any suitable *matching policy* can be defined by subclassing `ConfigurationPolicy`.
- Methods to be triggered *before and after every access* to a space can be specialized. This is useful for validating, modifying or rejecting entries, or for triggering *any useful side effect*.
- These hook methods and the matching algorithms can be plugged-in *dynamically* in a black-box fashion and take effect without restarting the system.
- A special *update method* can be triggered to automatically adapt affected entries whenever the policy in place is dynamically changed.

These properties allow a developer to implement any desired protocol over the tuple space communication. The protocol may be facilitated by charging low levels tasks to the space. This helps to increase efficiency and also reduce network traffic. The security of an application may be increased by performing consistency or authorization checks. Additional tasks may be triggered, like e.g. logging activities.

1.3.2 Heterogeneity

OpenSpaces is written on top of CORBA and offers access for *external clients*, such that any agents may use it for communicating and not only the ones provided with the framework. They may be written in *any language* for which an ORB is available. The minimal IDL module [2] makes it easy to implement the necessary protocol.

As a validation of the heterogeneity claim, a Java client has been developed, participating together with the standard clients in one of the example applications presented in chapter 5. This client runs on every *platform* for which Java is available, thus on all major operating systems.

1.4 Structure of the thesis

In chapter 2 we give an overview on Coordination and on LINDA, the first tuple space coordination model. Chapter 3 presents an extract of the evolution of tuple spaces, featuring milestones on the way towards configurable and heterogeneous models. In chapter 4 OPENSPACES is introduced, its core design, the important framework contracts, and its implementation. Chapter 5 features the case studies validating the claims of the model. In chapter 6 we discuss the issues of distribution and heterogeneity and present two more examples. In chapter 7, finally, we conclude.

[2] CORBA's Interface Description Language. See chapter 6 for details.

Chapter 2

Coordination and LINDA

2.1 Coordination

The success of tuple spaces originates from their advantageous model for communication amongst cooperating agents. The proposal of LINDA led to many activities in a new area of *Coordination* research. In this chapter we outline this research area, then we present LINDA, its features and a typical scenario of use, and discuss its significance as a coordination model.

2.1.1 Need for Coordination

With today's ubiquitous networks more and more large and complex distributed applications take advantage of the increased raw computational power provided by the cooperation of a multitude of connected computers. The crucial point in building such applications is to establish the necessary *communication structures* between the participating processes allowing them to synchronize and exchange data.

The cooperation of so many processes in such „virtual parallel computers" presents a new challenge to software engineering: how to *coordinate* large numbers of *concurrent activities*. The classical programming languages are based on extensions to the sequential programming paradigm, thus they are not very well suited to handle these problems. To exploit the potential of highly parallel systems we need special *programming models* that explicitly handle the *coordination* of concurrent activities.

With LINDA, the notion of a *coordination model* has been introduced (see below for a definition). The tuple space system has shown that computation tasks can be treated completely independent („orthogonal") of coordination activities [GC92]. Since its introduction a number of other models have been developed,

one important aim of each is to *abstract away and encapsulate technical details of the communication* between the cooperating components of the applications. This allows developers to concentrate on a higher level of abstraction, closer to the task of the application they build. Moreover, the facilitated communication structures provide for modular designs, the components are easier combinable and reusable.

2.1.2 Definitions

The precise definition, what exactly is meant with the notion of *coordination* differs to some degree with each model that has been devised. In an overview on the area of coordination by Malone and Crowston [MC94] we find a general definition: *Coordination is managing dependencies between activities.* These dependencies arise from the collaboration of the mentioned activities.

Carriero and Gelernter, the authors of LINDA, defined in [CG90]: *Coordination is the process of building programs by gluing together active pieces*. With ,,active piece" they refer to any process executing independently and concurrently to the other pieces. ,,Glued together", the cooperating ensemble of the processes forms the application. The ,,glue" is therefore responsible to provide the necessary means to communicate and synchronize: *A coordination language is the glue that binds separate activities into an ensemble*.

A coordination language is *the linguistic embodiment of a coordination model* [GC92]. LINDA, for example, defines a set of primitives that can be added to a (computational) host language. There are realizations on top of several host languages, like C-LINDA, and many more.

In general we can say that a coordination model defines how agents interact, how interactions are controlled. It offers communication and synchronization facilities in order to support cooperation amongst agents.

2.1.3 Coordination Models

LINDA is the first coordination model as such. It acts as glue between concurrent processes by offering the simple and uniform communication primitives that are easily integrated to tuple space protocols. Yet the tuple space model is powerful enough to express most of the common architectures in parallel programming [CG90].

LINDA has been used extensively in many applications (see [COORD96], [COORD97], [COORD99] for examples). Over the years a number of other similar models evolved, their main purpose being to address some of the deficiencies of the standard model. Issues such as distribution, security, additional operations, were added and optimized. (For an overview, see the next chapter.).

Apart from coordination models that extend and vary the concept of tuple spaces, several models were devised with different approaches. One classification, proposed by Papadopoulos and Arbab is *data- vs. control-driven* coordination models (see [PA98]). Data-driven models like tuple spaces focus on the data that is exchanged between the coordinated entities, whereas the control-driven models focus on the *flow* of control and the connections between the participants. Examples of control-driven models are *Manifold* [Arb96] or *ToolBus* [BK96].

2.2 LINDA

The concept of tuple spaces was initially presented 1985 by Carriero and Gelernter with the LINDA system [GC85]. It is essentially a blackboard architecture offering a central repository (a „blackboard") used to exchange information between agents [SG96].

LINDA introduced the paradigm of *generative communication* which overcame the tightly coupled structures of traditional communication with RPC or message passing. To communicate some information, the sender process generates a new data object (a *tuple*) and writes it to the shared memory space (the *tuple space*) from where the receiver can retrieve it. With this mechanism generative communication provides both spatial and temporal *decoupling* of the participants. They do not have to know their identities or locations, nor is necessary that they all exist at the same time.

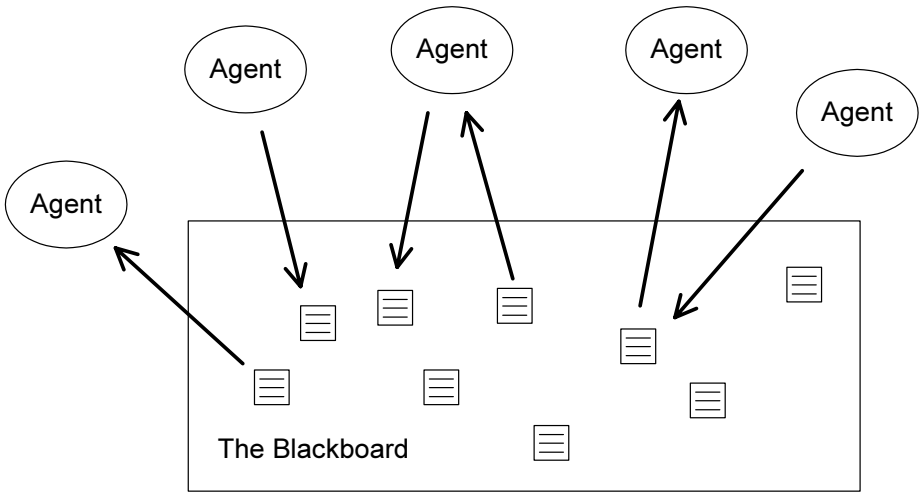

Figure 2.1: Agents communicating via a blackboard.

2.2.1 Primitives

The LINDA model defines a small set of operation primitives to access the tuple space: `out(t)` writes a tuple t to the space, `in(s)` takes a tuple from the space that *matches* the template `s`, `rd(s)` retrieves a copy of such a tuple without consuming it. The template tuple defines a mask - usually with a number of wildcards - which is used to lookup a tuple according to the matching rules (see next section). The retrieving operations suspend the receiver process until the desired tuple is found. Non-blocking variants of the retrieving operations are `inp(s)` and `rdp(s)` („predicate" variants). They immediately return with a matching tuple or an indication of failure. [1]

By providing a template tuple, the receiving process defines a constraint for the lookup of some tuple in the space. This associative lookup allows an agent to specify *what* should be retrieved without the need to know *where* to find it.

2.2.2 Matching

In LINDA a tuple is a sequence of typed fields. Wildcards in a template tuple have to be typed, too. The standard *matching algorithm* in original LINDA are as follows: a tuple matches a template if the corresponding value of a detected tuple will be bound, i.e. the retrieving operations in LINDA do not actually return a complete tuple, they access the (local) variables and set their values.

- Both have the same number of fields (arity)
- Corresponding fields have the same type
- The values of corresponding fields either have the same value or one of them is a wildcard.

In figure 2.2 some simple examples are given of the standard matching alogorithm in LINDA. Wildcards are denoted with a `?`-prefix to the variable to which the corresponding value of a detected tuple will be bound, i.e. the retrieving operations in LINDA do not actually return a complete tuple, they access the (local) variables and set their values.

[1] The naming of the operation primitives originates from LINDA's parallel computing context, where one global tuple space is used, a process gets `in`-put from and sends `out`-put to the tuple space. In a object-oriented and distributed context this seems not very intuitive, therefore in OPENSPACES we use the following naming convention, borrowed from Java Spaces: `write` for inserting entries to the space, `take` to remove them, and `read` to retrieve a copy.

```
{   // process A:
    // ...
    out("hello", "world", 123, 3.14);
    // ...
}
{   // process B:
    // ...
    int i, j; float f;
    rd("hello", "world", 123, ?f);     // succeeds and binds 3.14 to f
    rd("hello", "world", ?f);          // fails: wrong arity
    rd("hello", "world", ?i, ?j);      // fails: wrong type
    in("hello", "world", ?i, ?f);      // succeeds: binds values to i and f,
                                       // and removes the tuple
    rd("hello", "world", 123, ?f);     // fails: no tuple found anymore
}
```

Figure 2.2: Examples for the matching rules in C-LINDA

2.2.3 Active Tuples

LINDA offers a further operator, `eval(a)`, which puts an active tuple a into the tuple space, where it will be evaluated and end up as an ordinary (passive) tuple. The form of an active tuple is the same as of a passive one with the exception, that one or more fields are specified as a function `f(x)` where `f` is a (globally known) function and `x` the provided parameter. The field will finally be replaced with the result of the function, and the (now passive) tuple is put into the space. For instance, the call `eval("square",a,sq(a))`, with `a = 3` and the definition `sq(x) := x * x`, will finally produce the tuple `("square",3,9)` in the space.

Since the used functions must be known to the space, the `eval` operation is mainly interesting in parallel environments. In a distributed system there are some limitations. Either the operation is restricted to a set of functions a priori known by the space, or their code has to be provided by the client, which implies security related problems.

However, the interesting aspect of a tuple space is founded in its use for coordinating agents, the significance of the `eval` operation is actually closer to being a handy shortcut. Hence, this operator was often left out in later LINDA implementations and extensions to the tuple space model. The operation has not been included with OPENSPACES, neither.

2.2.4 Usage scenario

As an example of a typical scenario of using tuple spaces, figure 2.3 shows a diagram of a simple eMail–system [Cia99]. The example uses multiple spaces, which have been introduced in [Gel89]. Compared to a global space which is accessible by all agents, such a setup offers more control of privacy. The ,,Post Office Tuple Space" is known to every participating agent, the ,,Receiver *X* Tuple Space" only to its owner *X*. (See 3.2.1 for discussion of multiple spaces.)

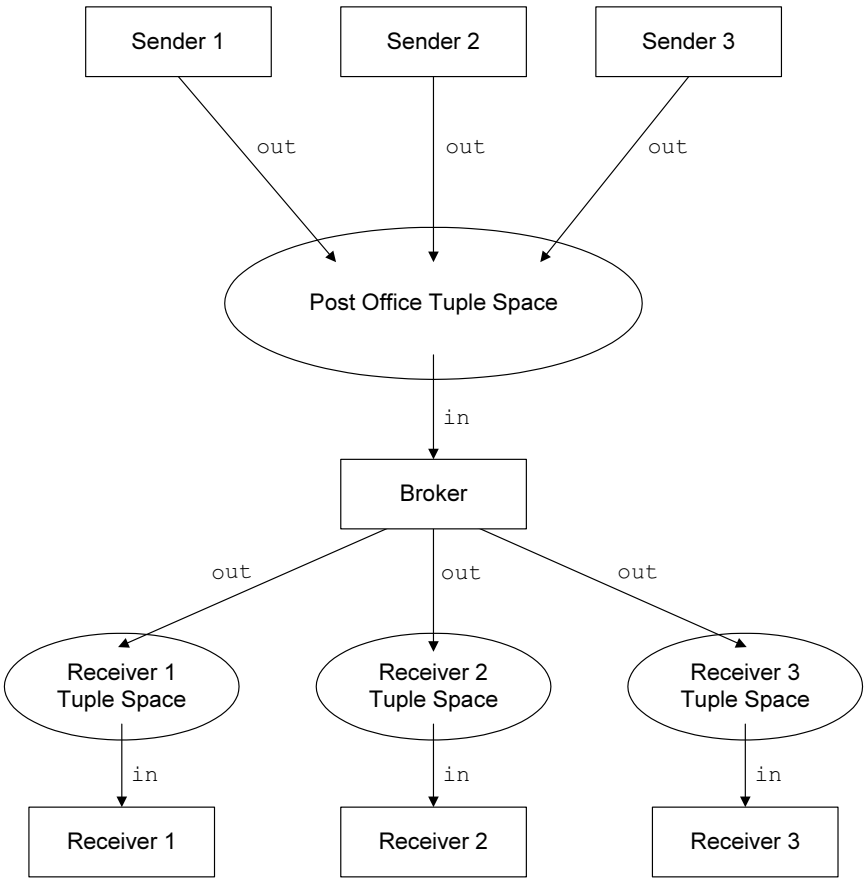

Figure 2.3: A usage scenario for a simple mail system.

Each message tuple has at least fields for the names of the sender and receiver and for the message body. The sender agents put the message tuples to the ,,post office" space. The broker agent continuously collects all messages and distributes them to the ,,mailbox" spaces of the receivers. Each mail client checks its own space for incoming mail.

Discussion. An even more simple mail system could be designed with a single global tuple space. The space just acts as repository, the clients check if any tuples are present with their name in the receiver field and collect them. In such a setup it is, however, up to the clients to only access mails for which they are the designated receiver. There is no control, if a client retrieves other messages by using other agent's names.

Hence the example uses private spaces for each client which are accessible only by the owner and the broker agent. The post office space may be used by anybody, it wold however make sense, if it would be *write only* for ordinary users.

2.3 LINDA as a Coordination Model

After the more technical details about the LINDA tuple space, we continue with a discussion of its significance as a coordination medium.

2.3.1 Advantages

Simple yet Expressive. Tuple spaces capture both synchronization and communication in a simple and natural way. The minimal set of operations is sufficient to perform this. Tuples can represent any information, at the same time they can be used for synchronization.

Many problems map in a natural way to tuple space communication. Despite its simplicity it has been shown in [CG90] that LINDA is capable of expressing a large class of parallel and distributed algorithms. Architectures like *Master-Worker*, *Specialist Parallelism*, *Pipes and Filters*, and more [SG96], are easily adaptable to tuple space communication.

With optimized implementations it has been proven, that the a tuple space system can be implemented to be as efficient as RPC or message passing. The model has successfully been used for many applications.

Decoupling of the Participants. While LINDA has been introduced as a tool to supporting the development of parallel applications, it soon became clear that its decoupling properties were highly attractive for open distributed computing. An agent can deposit a tuple before the actual receiver is even running and without knowing where exactly it is. These properties are crucial for flexible integration of components spread over the network, they considerably facilitate setup and configuration of dynamically joining participants.

Associative Addressing. By providing template tuples agents specify *what* information they are interested in, and not *where* to find it. This associative retrieval

is more abstract than specifying a certain message by its number or address and helps to abstract away details of the communication structure.

Asynchronous Communication of Concurrent Agents. Tuple space communication is implicitly asynchronous, tuples are picked up at the moment when the receiver is ready to do so. Concurrently running agents can use the space to synchronize, but their thread of execution remains independent of each other.

Separation of Concern. A concrete LINDA implementation is an extension to an existing language. The small set of operators are completely independent of the other constructs of the host language. Therefore it is possible to implement LINDA variants for almost any programming language. There are implementations in virtually all major languages, e.g. C, Pascal, Prolog, Lisp, Java, and many more.

The fact that inter-agent communication can be treated with a set of operations which is completely independent from the rest of the host language shows clearly, that computation can be treated strictly *orthogonal* to coordination. This helps to separate the code concerning the two domains within an application.

2.3.2 Limitations

In spite of its advantageous characteristics as a coordination model, the original LINDA suffers from some drawbacks. To be used in open distributed systems the following problems must be solved:

Distribution. LINDA was developed in the context of high speed parallel programming and offers no means for distribution.

Privacy. There is only one global space. As we have seen in the Post Office example in 2.2.4, it is often desirable to have a finer grained *namespace*. Multiple tuple spaces offer a simple solution for preventing accidental or malicious interference.

Security. There are no security features like authentication of access or encryption of the transmitted and stored data.

Multiple Read Problem. LINDA's associative read operation is non deterministic, i.e. a process can not perform a series of read's to finally get *all* tuples matching its template (see 3.2.4 for detailed discussion).

Fault Tolerance. In a distributed environment, it is crucial to handle failures originating from connection interrupts, slow reactions, and the like. Also, clients may crash before completing their transactions. Without any prevention, it is likely for a space to get into in an inconsistent state and become unusable.

Optimizations and Abstractions. Tuple space protocols are often a bit awkward. They typically include constructs like, e.g., counter-tuples which agents have to increment by performing the sequence of `read` – update – `write`.LINDA offers no means to abstract such ,,boiler plate code" away, which would help to make the protocol easier, less prone to errors and inconsistency, and better maintainable and extensible. The need for handling such low-level tasks in the client code also contributes to the trend of spreading coordination code over different parts of an application, which annihilates a good part of the benefits of the separation of concern.

Many extensions to the original LINDA model have been proposed in order to overcome some of these limitations and to deal with the requirements of different application areas. We continue by presenting a selection of milestones in the tuple space evolution, each of which introduced new approaches and solutions. In particular we focus on the line towards OPENSPACES, i.e. models integrating features like distribution, object-orientation, configurability, and heterogeneity.

Chapter 3

Evolution of the Tuple Space Model

Tuple spaces have been amongst the most successful coordination models. For this thesis we intended to extend the model to make it *configurable in its behavior* and at the same time *accessible for heterogeneous agents*. We therefore focus in this chapter on the evolution of the tuple space model along the line leading towards these properties and present the most important milestones. We have to leave out other models proposing innovations beyond this domain, for an overview, see [PA98], [Cia99], or [OTZK01].

After an overview of the most important properties of the investigated models we present two prominent systems that introduced general extensions which have been adapted by many subsequent tuple space models. Then we have a closer look to object-oriented implementations. This programming paradigm offers a high degree of extensibility, which is an important requirement in dynamically changing systems. Finally we discuss the systems which offer some options to modify the behavior of the model.

3.1 Overview

We selected a set of important properties that characterize a tuple space coordination model:

- *Distribution:* Is a tuple space accessible over the network?
- *Multiple Spaces:* Is there only one global space abstraction, or can several spaces be employed?
- *Bulk Retrieving:* Are operations realized for retrieving more than one single tuple? (See discussion of the multiple read problem in and 3.2.4.)
- *Active Entries:* Is there some kind of execution model for processes that are initiated by writing tuples to the space à la LINDA's `eval` operation?

- *Object-Orientation:* Does the implementation use OO-technology?
- *Event Notification:* Is there an event notification mechanism to notify clients of arriving tuples?
- *Access Control:* Can either unauthorized clients be rejected, or can data objects be validated before being written to the space?
- *Customizable Matching:* Is it possible to employ alternative matching algorithms?
- *Configurable Behavior:* Can the semantics of the model be modified or changed?
- *External Clients:* Can the model be used by external clients without the need of first writing wrappers or libraries?

Table 3.1 shows the overview on the coordination models and their properties. A symbol „$\surd$" in a field thereby means that the model on the same row disposes over the feature indicated by the column, „-" means absence of the feature, a number refers to a remark listed below.

3.2 Extensions to LINDA

The basic Linda model has been extended in various ways by different languages and architectures in order to deal with the requirements of different application areas. These shifted from high speed parallel computing over distributed applications to Internet-based multi-agent architectures. Two important changes to the model – namely multiple spaces and extended operation primitives – have soon been introduced and were adapted by later proposals.

3.2.1 Multiple Tuple Spaces and Distribution

LINDA had been proposed as a tool for developing programs on closed parallel computers where the single global tuple space is used by a controllable set of processes. As soon as it was beginning to be exploited as a coordination model in distributed environments, the need for a finer grained structure of repositories arose. If too many programs may use the same tuple space, there are good chances that interferences may occur, either maliciously or accidently. [1]

[1] In principle a single tuple space could be distributed over the network. There are several strategies how this could be organized, from a centralized tuple server to partial or total replication (see e.g. [Gel89] for discussion). All variants, however, are costly in the least, and the idea has not gained much support so far. Hence, when calling a tuple space model „distributed" we refer only to remote accessibility of the spaces.

Coordination Model	Property									
	Distribution	Multiple Spaces	Bulk Retrieving	Active Entries	Object-Orientation	Event Notification	Access Control	Customizable Matching	Configurable Behavior	External Clients
LINDA (1)	-	-	-	√	-	-	-	-	-	-
PARADISE	√	√	-	-	-	-	√	-	-	-
BONITA	√	√	√	-	-	-	-	-	-	-
MATSUOKA & KAWAI	√	√	√	-	√	-	-	(2)	(3)	-
OBJECTIVE LINDA	√	√	√	√	√	-	√	(2)	(3)	√
BERLINDA	-	√	-	-	√	-	-	(2)	(3)	-
JADA	√	√	√	-	√	-	-	(2)	(3)	-
JAVASPACES	√	√	-	-	√	√	-	(2)	(3)	√
TSPACES	√	√	√	-	√	√	√	(2)	(4)	-
LAW-GOVERNED LINDA	-	-	-	-	-	-	√	-	√	-
PRG.COORD.MEDIA	√	√	-	-	-	-	√	-	√	-
LUCE	√	√	-	-	(5)	-	√	-	√	(5)
OPENSPACES	√	√	√	-	√	√	(6)	√	√	√

Table 3.1: Overview of the investigated properties

Remarks

1. With LINDA we refer to C-Linda, the first implementation.
2. & 3. Statically through subclassing.
4. Subclassing plus dynamic activation of new code.
5. A Java client with provided Prolog engine is included.
6. No direct control of client identity, password protection realizable with policies.

Of course, the object-oriented systems can be extended to support more features than the presented standard versions.

Multiple tuple spaces have been introduced as an effective way of hiding information [Gel89]. Information within a tuple space can only be accessed by those processes that know about the tuple space. In the post office example in 2.2.4

the private space of each client is not known to other clients and therefore not accessible.

Designs of multiple space models have to take account of two issues, namely if the spaces are first class objects, and what relationship should be between the tuple spaces. Spaces as first class objects can be used as any other first class value, thus be inserted into other spaces. This approach rises complex semantical questions, e.g. how to handle a write operation on a space that has been consumed by another agent. Hence most models do not support first class spaces. The tuple spaces can either be unrelated or they can be related creating some form of hierarchy[Hup90]. Most implementations use only a flat structure.

Using a tuple space system in a distributed environment requires some way of accessing a remote space for the agents. An implementation needs to provide some communication facility and also a mechanism to get a handle to a certain space. In addition the communication over a network is less reliable than inside a closed parallel system. To handle this issue is another challenge for a distributed tuple space model.

We now present two examples of models that both feature multiple spaces.

3.2.2 PARADISE

PARADISE [Sci95] is a classic LINDA implementation featuring multiple tuple spaces and distribution. It includes the standard LINDA operations - without `eval` - plus specialized operators to open and close a space and to define access restrictions.

The syntax of the standard primitives is extended with the indication of the space to operate on, e.g. `rd@mySpace(template)`, where `mySpace` is a handle to the specified space. The space handle holds an access key defining a set of restrictions, the owner may pass it to other agents it trusts.

PARADISE is implemented over two host languages, C and Fortran and it runs on networks of heterogeneous workstations (Sun and Intel). The matching of tuple values is complicated by the different platforms, PARADISE defines data type equivalence tables for this end.

3.2.3 BONITA

BONITA [RW97] is a set of coordination primitives that enhance the functionality of the basic LINDA model and improve the performance in the context of distribution. BONITA supports *multiple tuple spaces*, each operation has an additional parameter indicating the space on which to operate. There are also operators that move or copy tuples between two spaces.

The model introduces a finer grained notion of tuple retrieval. The retrieving operations are split into a *request* for reading or taking and a *check*, if a tuple for the corresponding request has arrived. This allows a client implementation to issue several requests in parallel with less overhead, while in original LINDA each request has to complete before the next can be issued.

Moreover the handling of a blocking request becomes ,,network proof". If the network connection fails during a normal blocking request, the client could be suspended forever. With the divided approach of BONITA this can be detected if the `check` method does not immediately return, either successful or not.

3.2.4 Multiple Read Problem

Since tuple retrieval in LINDA is non-deterministic, it is not possible for a client to read *all* tuples at a space that match a given template. After a first read, the next attempt will probably return the same tuple again or any other, there is no guaranteed order. One workaround for this problem is to supply an unique index to each tuple that is written to the space which can then be used in the template (see chapter 5 for an example of this construct and its implications).

It would be desirable however to have a more general solution. BONITA [RW96] introduced for this end a new primitive called `copy-collect(ts1, ts2, template)` which moves copies of *all* matching tuples found at tuple space `ts1` to tuple space `ts2`. The agent can then `take` them one by one from `ts2`.

3.3 Object-Orientation

The object-oriented programming paradigm promises a maximum of *extensibility, reusability, modularity*. We therefore are mainly interested in coordination models supporting it.

Since objects encapsulate both state and behavior, an object-oriented LINDA implementation has to consider how to deal with objects put in a space. Most implementations however restrict them to be passive, to act merely as data containers.

3.3.1 Matsuoka and Kawai

The first proposal of an object-oriented LINDA system is described in a paper by Matsuoka and Kawai [MK88]. It is a distributed implementation in Smalltalk, introducing tuples and tuple spaces as objects.

Matsuoka and Kawai introduced a symmetrical mechanism for tuple space communication: both the sender and the receiver put a tuple into the space - the

receiver's tuple has the role of the template - at the space they are matched against each other and the receiver tuple's formal fields are instantiated with the sender tuple's values, then the receiver tuple is delivered to the receiver. In this way both kinds of tuples can be passed amongst agents and can be used without complete knowledge and understanding of their contents. In LINDA this is not possible, the receiver tuples even have to be hard coded in the receiver's program. The new symmetrical mechanism - Kawai and Matsuoka call it *orthogonal communication* - offers data encapsulation and possibilities to delegate tasks.

The models features multiple tuple spaces which are first class objects, they can be passed as elements of a tuple. As a novelty a *blocking write operation* is introduced, which suspends the sender process until the tuple is read or taken by another process. Innovations are also bulk retrieving primitives and operation variants that allow an agent to perform insertions or retrievals at more than one tuple space at the same time.

3.3.2 OBJECTIVE LINDA

OBJECTIVE LINDA [Kie97] is a LINDA extension aiming at distributed open systems. It supports multiple spaces and enhanced operations having each a timeout value denoting a duration after which the operation terminates and returns, successful or not. The `out` and `eval` operations return a boolean indicating their success. The `in` and `rd` operations take additional min and max values which are the boundaries of the number of objects they should return in a multiset.

OBJECTIVE LINDA introduces an *Object Interchange Language* (OIL) which is a programming language independent notation for describing objects. The operations of the coordination model are based on the OIL specifications of the (template-)objects. The matching algorithm has to be defined in the OIL definition of an object written to the space.

Furthermore, the language supports hierarchies of multiple object spaces and ability for objects to communicate via several object spaces. Object spaces are accessible through handles called *logicals*, which also give opportunity to define access restrictions. OBJECTIVE LINDA supports the *eval* operation in an object-oriented way: an object is moved to the space where it is activated.

3.3.3 BERLINDA

BERLINDA [Tol97] is a small generic framework written with the programming language Java, which can be extended to implement several LINDA-like coordination languages. It runs only local on a single Java VM.

BERLINDA defines four base classes: `Agent` for the clients, `Medium` to hold a shared memory abstraction, `Element` for the objects to be exchanged and

`Signature`. A signature can be carried by an element and provide meta information like a type description of the element or the receiver address. Elements define the desired matching algorithm as a member function.

3.3.4 JADA

JADA [CR96] is a development of the University Bologna combining **Ja**va with Lin**da**. It offers coordination of multiple threads on a single Virtual Machine or distributed over the net. The choice of Java allows client of a space to run in a WWW browser as an applet.

JADA provides for multiple spaces, bulk primitives and timeouts for the retrieving operations. Tuples are sets of objects, formal values are expressed with the Java meta class representation of the specified type. E.g. with the definitions `Tuple a = new Tuple("test");` and `Tuple b = new Tuple(new String().getClass());` the tuple `a` would match the template `b`.

The matching is based either on the standard Java `equals` method or - if the object implements the interface `JadaObject` - on the ad hoc `matches` method which can be used to customize the matching algorithm.

To access a tuple space over the network a `TupleClient` connects to a `TupleServer` object that forwards requests to its space. The `TupleClient` has to specify the IP address and portnumber of the `TupleServer` to connect, the communication uses sockets, in later versions RMI.

3.3.5 JAVASPACES

JAVASPACES [FHA99] is another Java model, developed at SUN Microsystems as part of the Jini architecture, featuring distributed multiple tuple spaces.

Tuples are instances of classes implementing the interface `Entry`, they are stored at a `JavaSpace` in their *serialized form*. Therefore the matching relies on this form: two objects match, if their serialized forms match. To specify wildcards the Java keyword `null` is used. An entry `e` matches an template entry `t`, if `e` belongs to the same class as `t` or to a subclass and if the serialized values either are equal or one of them is `null`.

The `write` operation returns a `Lease` object for each entry written that specifies the duration after which the entry will be garbage-collected and removed from the space. The retrieving operations `take` and `read` use a timeout parameter after which the call will return, if it was not successful.

Each access operation uses a `Transaction` parameter, which can be used to group a series of accesses and make them atomic, guaranteeing that either all of the operations complete or none.

As a speciality, JAVASPACES introduces a `notify` method. A client can register with it to be notified if an entry has been written to the space that matches

the supplied template. This mechanism provides a synchronous communication style without blocking the process.

3.3.6 TSPACES

TSPACES [WMLF99] is a Java tuple space implementation of IBM Research Division with a strong focus on database technology. It runs distributed with a proprietary protocol on top of TCP/IP and offers multiple flat spaces. Tuples are subtypes of the abstract `SuperTuple` class, their fields can be any serializable Java objects.

Three variants of matching strategies are available: *subtype matching*, *named fields* and *queries*. The first is the same as in JAVASPACES. The second approach can be used for tuples having named fields: fields are compared according to their labels, their position is ignored. (We use the same approach in our Market Place example application in chapter 5). The third option is possible through the fact that tuple spaces are backed up in databases. A query–matching performs an index–scan and returns a set with all tuples satisfying the restriction of the specified query. Fine grained and/or combinations are possible, similar to SQL's `select` operation.

Clients find a tuple space by specifying the host address and port number of the corresponding server. The standard LINDA operations are provided, bulk retrieving operations, too. A new rendez-vous operator `rhonda`, which takes a tuple and a template as arguments blocks the sender until another client performs a `rhonda` with a matching tuple/template pair which will be exchanged with the senders pair.

TSPACES provides transactions and event notification. It also supports a simple user authentication protocol and access control options. A special mechanism in the TSPACES API allows a programmer to define new space operations, whose code can then be activated dynamically by uploading it to the server.

3.4 Programmability

Whereas the presented object-oriented systems are statically configurable through subclassing or by the option to specialize the matching algorithms used, the last category of models we discuss here, have added *modifiable behavior* to the spaces. The first is LAW-GOVERNED LINDA (in 3.4.1) which introduced a *reaction model* allowing the space to intercept accesses. The work on PROGRAMMABLE COORDINATION MEDIA (3.4.2) introduced the notion of programmability, extending the reactivity model. Recently LUCE (3.4.3) has been promoted, based on the same reaction model.

3.4.1 LAW-GOVERNED LINDA

In LINDA many consistency and safety requirements are based on *voluntary conventions*: an application has no control over the behavior of external clients. In open systems this can be fatal. LAW-GOVERNED LINDA [ML95] adds a mechanism to *enforce laws* for tuple space communication.

A LAW-GOVERNED LINDA system is defined by a 5-tuple $< C, P, CS, L, E >$ where *C* is the communication *medium*: a LINDA tuple space, *P* is the set of *processes* or agents using the medium, *CS* is a set of *Control States*, each of them holding information about its associated process, *L* is a *Global Law* which defines *rules* that govern the interactions that take place at the communication medium, and *E* is a set of *controllers* that *enforce* the law. Each controller is associated to one process and its control state, and it has a local copy of the law.

The rules are defined in Prolog and can intercept the tuple space accesses. They can react to two kinds of events: *invocation events* occur on attempts to perform an operation, a *selection event* occurs when a matching tuple is found at the space. These events trigger a search for a rule that unifies with the event/tuple pair involved. The following operations can be used to define suitable reactions: `complete` actually carries out the invoked operation, used in conjunction with an `out` the tuple will be written, with `in` or `rd` the search for a matching tuple is started; `complete(t')` executes the operation with the alternative argument `t'`; `return` passes a detected matching tuple to the client agent; `return(t')` replaces the found tuple with `t'`; `out(t)` is the conventional LINDA operation, that may be used by a rule, the `in` and `rd` operations are not allowed to prevent the controllers from stalling; `remove` disconnects a process from the system.

With these operations and a few additional attributes, each application can formulate a suitable law to enhance the basic functionality of LINDA. An example of a simple law is shown in figure 3.1. It implements secure message passing by guaranteeing that message tuples of the form `[msg,from(s),to(r),contents]` can only be read by the correct receiver. The field `msg` thereby is a tag identifying the tuple as message, `s` is the sender's process ID, `r` the receiver's, and `content` is the actual message to be exchanged.

```
R1. out([msg,from(Self),to(_)|_]) :- do(complete).
R2. in([msg,from(_),to(Self)|_]) :- do(complete)::do(return).
R3. out([X|_]) :- not(X=msg),do(complete).
R4. in/rd([X|_]) :-  not(X=msg),do(complete)::do(return).
```

Figure 3.1: An example of a law guaranteeing secure message passing.

The variable `Self` is bound to the home-process, i.e. the process to which the controller is associated. The rule `R1` allows processes to `out` message tuples,

but enforces that the `from` field has the correct sender ID and can not be forged. `R2` allows a process to `in` a message if and only if its own ID is in the receiver field. Rules `R3` and `R4` provide standard LINDA treatment for any other tuples not having the `msg` tag in their first field and thus do not need to adhere to the above law.

Several kinds of security policies can be implemented with the LAW-GOVERNED LINDA model. A drawback is the global character of the law which is copied to every controller and therefore not easily reconfigurable.

3.4.2 PROGRAMMABLE COORDINATION MEDIA

In PROGRAMMABLE COORDINATION MEDIA [DNO97] the notion of *programmability* is introduced to tuple space systems. The work exploits the benefits of a coordination medium with extensible and modifiable behavior. It uses an enhanced version of the Prolog tuple space system *Agent Communication through Logic Theories* (ACLT) as a validation [ODN95].

In ACLT tuples are expressed with logic terms which enables agents to reason over the state of the space. A LINDA read operation means to provide a term (the template) which the system tries to unify with the theory represented by the union of all tuple terms at the space. For PROGRAMMABLE COORDINATION MEDIA a mechanism has been added to enable the space to *react* to communication events, i.e. to access operations.

A general set of operations allows the programmer to define the reactions to events in the form `reaction(Event,Body)`. The `Body` of the reaction thereby is the conjunction of a set of reaction goals. These are defined using state primitives (`current_agent,current_op`), term predicates (`equal`,`unifiable`), or communication primitives (`out_r,in_r,rd_r`) - the postfixes indicate the variants of the usual primitives, that can be triggered by the reactions. Blocking operations are not allowed.

As a simple example of a reaction, the following definition causes the tuple space to react to an `out` operation by trying to remove two `p/1` tuples, of which one should be a `p(a)` and, if they are found, to replace them with a single `pp/2` tuple.

```
reaction( out(p(_)),
          ( in_r(p(a)), in_r(p(X)), out_r(pp(a,X)) ) )
```

The presented model is very generic and allows many behavior policies to be programmed. In the presented version the system is however restricted to Prolog and there is no means to use it with external agents.

3.4.3 LuCe

Logic T**u**ple **Ce**nters, or LuCe [DO99], is a descendant of the above mentioned ACLT and Programmable Coordination Media systems. It has a focus on multi agent systems spread over the Internet.

A *Tuple Center* is a logic tuple space implementation with programmable behavior similar to the Programmable Coordination Media system. The reactions of the coordination medium to communication events can be specified using *ReSpecT*, a superset of the operations defined in ACLT.

In LuCe the agents are allowed to (re-)program a space by writing ReSpecT tuples. The space is conceptually divided in two parts: the tuple space used for ordinary communication tuples, and the *specification space* containing the reaction clauses in the form of (logic) tuples that specify the behavior of the system.

For Internet usability LuCe integrates Java agents by providing a light weight Prolog engine written in Java. The system is particularly suited for the construction of multi-agent systems involving autonomous, pro-active agents.

In [COORD00] TuCSoN has been presented, a direct heir of LuCe, adding distributed *access control* to the combination of reactive logic tuple spaces and the ReSpecT specification language [COZ00]. The tuple centers reside each on a Internet node, clients call operations at remote spaces in the form `ts@node?op(tuple)`. At each node the permissions are set in an access control list per Tuple Center, stating the granted permission for registered agents.

3.4.4 Drawbacks of existing approaches

The presented systems offer many facets of tuple space coordination. The object-oriented systems can be extended by subclassing to add new functionality, but they are more or less static in their behavior.

In the „programmable league" there are models with sophisticated mechanisms to provide reactivity and modifiable semantics. They, however, are bound to some limitations:

- *Internal agents.* The models only are usable with their own provided internal standard clients. None of the systems enables arbitrary clients to access it. A true generic communication medium must offer access for any kind of participant capable of performing the required protocol.

- *Client Implementation Languages.* The presented reactive models adhere to logic programming as implementation language for the clients. The only exception, LuCe, offers access for Java agents using the provided Java-Prolog engine. Other languages are not supported. As shown for example

in [PA98][2], for modern complex systems it must be possible to realize each component in the programming environment that is best suited for its task.

- *Programming the Medium.* In all „programmable" systems the spaces' behaviors can only be specified in Prolog-like dialects. While in principle everything can be expressed, many desirable actions become highly complicated to implement. Needs of real-world applications like file or database access, complex computations for encryption, and others, are much easier realized with other languages. Besides, the usage of Prolog in commercial applications is not very common, neither exist many programmers familiar enough with the language to integrate logic components with their software.

- *No Object-Orientation.* It would be desirable to have a configurable system exploiting the benefits of object-orientation, which is not without reason today's preferred programming paradigm.

In the next chapter we introduce OPENSPACES, starting with a description of the design of the framework. Then we discuss its main contracts and their implementation.

[2] p. 3: „Gradually, it became apparent that no unique programming language or machine architecture was able to deal in a satisfactory way with all the facets of developing a complex and multifunctional application."

Chapter 4

The OPENSPACES Framework

OPENSPACES is an object-oriented framework, written in Smalltalk, which offers the possibility to implement a variety of data-driven coordination languages. The semantics of instantiations of the framework may have characteristics of several existing models. In addition OPENSPACES offers fine-grained configuration options that can modify the standard behavior of the medium. They actually can be used to charge the medium with additional responsibilities. Moreover, the configurations may be dynamically changed without restarting the system.

The main features of OPENSPACES are:

- Standard operations plus bulk retrieving
- Multiple distributed spaces
- Event notification
- **Configurability** of behavior of the spaces
- Internal and **heterogeneous** external agents

In this chapter we present the basic features and their realization through design and implementation of the framework.

4.1 Overview

The core defines a blackboard style medium, implemented in the class `OpenSpace`, which allows agents to interact by exchanging data encapsulated as `Entry` objects. The provided standard agents are specializations of `SpaceAgent`.

To get a reference to access an `OpenSpace`, an agent either calls the globally accessible `SpaceServer`, a name server that maintains a collection of all currently registered spaces, or it retrieves the reference from an IOR (*Interoperable Object Reference*, see chapter 6).

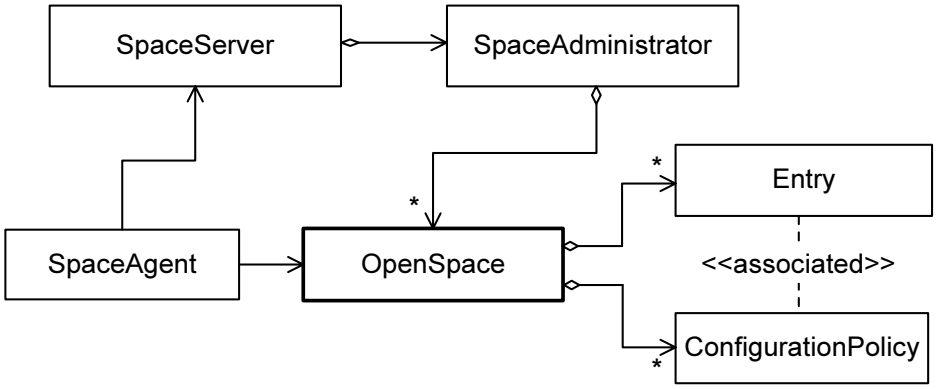

Figure 4.1: Overview on the structural relationships of the core of OPENSPACES.

The space offers the standard accessing primitives: `write` inserts an entry into the space, `read` and `take` retrieve an entry by associative lookup.[1] A *template* is an entry that is used as a mask to look up an entry from the space. The activated *matching algorithm* determines whether or not an actual entry matches the template and may be returned for it.

In OPENSPACES the matching algorithm is customizable. It can be adapted to the needs of an application. For every specific subclass of class `Entry` there may be a *particular matching algorithm*. Each `Entry` subclass has to be associated with a `Configuration Policy`, which defines its matching strategy. In addition, the policy object *controls the access to the space* for entries of the associated class. This is realized with validating methods that are applied before and after all accesses. These methods may basically trigger any actions.

4.2 The Core Classes and their Relationships

We now present the core classes shown in figure 4.1 and 4.2 and their structural relationships and collaborations.

[1] For the naming convention see footnote in section 2.2

4.2.1 Entries

Entries contain the data which space agents may exchange via a space. `Entry` is the abstract root class for all space entries. It has no attributes, applications must define subclasses with the necessary attributes needed to exchange specific information between the participating agents. Subclasses may add an arbitrary number of instance variables to hold any values.

The class of a concrete `Entry` descendant is the key used to associate the type of entry with a `ConfigurationPolicy` governing the space's behavior when accessed with instances of the specified entry class.

Depending on the needs of an application, single variables or collections may be used to hold the data to be communicated. The `Form` from our market example (presented in chapter 5) uses dictionaries, the `Tail` entries have two single instance variables. There is no restriction in combining both techniques, as long as there is an adequate declaration for the entry class in the IDL (see chapter 6)

The templates used for associative retrieval are normal entries, usually of the same class as the searched data object. They can have 0..n fields marked as wildcards using `nil` values. The actual semantics of the matching have to be defined in the configuration policy associated to each entry class.

There is no support for active entries in OPENSPACES, the configuration policies, however, may perform certain tasks on the computer hosting the space.

4.2.2 Configuration Policies

The configuration policies represent the semantics of the space's access operations. The class of each entry that is to be used, must be *registered* at the space with a corresponding policy object. Attempts to access a space with entries of unregistered classes are rejected.

The class `ConfigurationPolicy` defines the *matching algorithm* that will be used for the retrieving operations with the associated entries. It additionally has a set of *access control methods* which are applied before and after each access operation with the involved entry types. The use of class `ConfigurationPolicy` for setting up specific behaviors of a space is discussed in depth in section 4.3.

4.2.3 Spaces

The space abstraction `OpenSpace` represents the blackboard medium. It holds a collection of entries and offers several ways of accessing it. The standard primitives adapted from LINDA are supported.

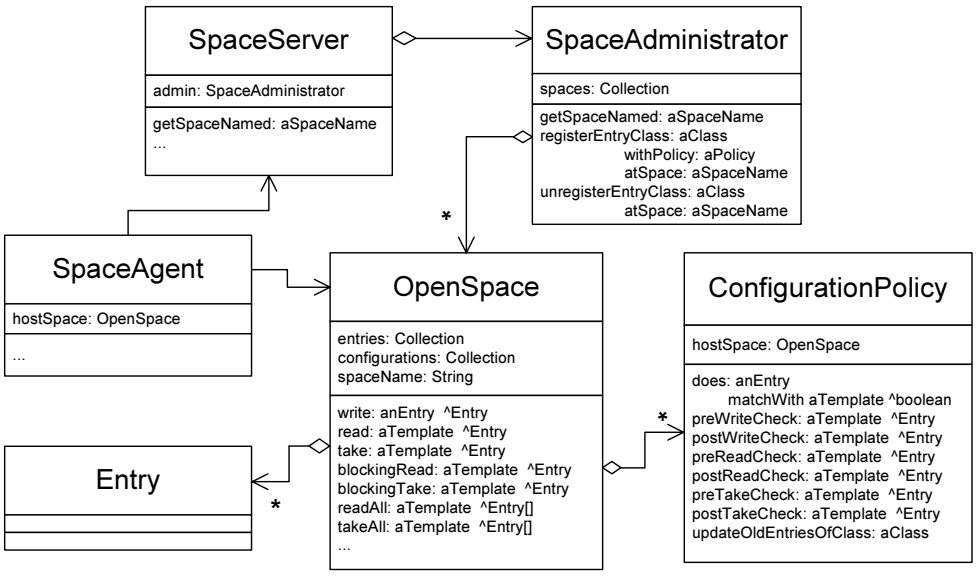

Figure 4.2: The core classes with their main attributes and methods.

The two retrieving operations `read` and `take` use both a template entry as parameters, which is used for an associative lookup of a matching entry. The template is an `Entry` which may have some of its data fields defined and some not. Using a typical matching strategy the undefined fields act as wildcards for the lookup, those with actual values restrict the selection of an entry.

The lookup in general is non-deterministic, i.e. if there are more than one entry at the space that would match a template provided by a retrieving operation, there is no defined order which one will be delivered.

In contrast to LINDA, the operation primitives actually move entry objects from and to the space. This fits the object-oriented approach of encapsulation better and is actually much easier to realize in a distributed context. The redundant returning of those values provided with the template for retrieving operations is neglectable concerning network load: in typical use cases they just represent some names or numbers.

The simple `read` and `take` operations are non-blocking, i.e. the calls return immediately, either with a found entry or, if nothing has been detected, with a `nil`-value. OPENSPACES also supports the blocking variants: `blockingRead` and `blockingTake`. These cause the calling client process to suspend until a matching entry is available. Each blocking read or take operation for which there is not immediately a matching entry is handled with a (lightweight) process at the space which makes a *reservation* in a list that is checked at each write access. If a matching entry is written, this process is resumed, it executes the retrieving

operation and returns its result to the client, which can then continue.

Two additional bulk-retrieving operations are supported: both `readAll` and `takeAll` act the same as their single counterparts with the exception that they return collections with *all* currently available matching entries. They are useful in some situations and help to avoid the „multiple read problem" (Refer to section 3.2.4).

The exact behavior of an `OpenSpace` may be different for any class of used entries. Indeed, part of the space's behavior is encapsulated in the *configuration policy* objects, of which the space holds one for each registered entry class. Therefore the `OpenSpace` class provides the functionality to manage the *mapping* between entry classes and the configuration policies which are to be used with all instances of them. Note that attempts to use unregistered entries types are rejected.

4.2.4 Space Agents

A space agent [2] is the provided standard user abstraction for the space. It has a reference to its current host space, which it gets from the space server. Depending on the needs of an application, the agent may switch to other spaces, or hold references to several space.

The class `SpaceAgent` is often subclassed to add application specific behavior and hide the underlying communication structures. For an example refer to the chapter on the Case Studies, figure 5.6.

4.2.5 The Space Server and Its Administrator

As mentioned OPENSPACES supports multiple spaces to be used concurrently. To become accessible for agents they are registered with the central `SpaceServer`, who manages their accessibility and life cycle.

The space server can act as a factory offering a method to create new spaces. It holds references of all currently available spaces and offers a means to agents to lookup a space by its name. A reference to the space server is held by the CORBA Naming Server which every client can access (see Section 6). The class `SpaceAdministrator` is the actual delegee holding the registered spaces and performing the lookup by name.

A second variant of getting a space reference is the use of an IOR, a special string holding the necessary information to reconstruct a remote reference (see chapter 6). This string is written to a file and can easily be made accessible.

[2] The name is *NOT* inspired by Star Trek!

4.3 The Access Contracts

After having briefly described the core classes, we now present how they interact. An important part of the framework's flexibility originates from the contracts of the different access methods available to clients of an `OpenSpace`. We describe in detail the read contract. The take contract is analogous. The write contract differs in some parts.

4.3.1 Intercepting the Access Operations

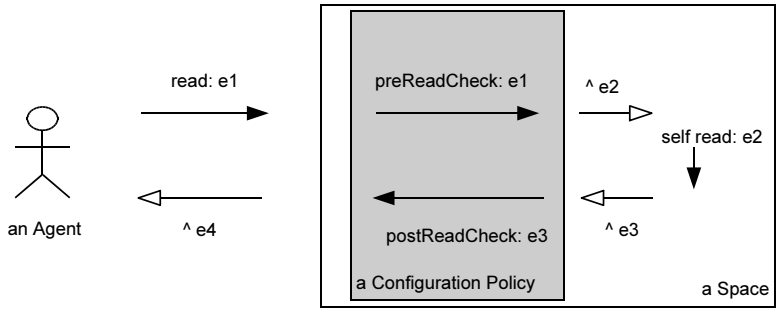

Figure 4.3: The configuration policy intercepts every `read` operation before and after the actual space access.

The mechanism providing full control over each space access intercepts the access operations and exchanges the used entries with potentially modified copies. Figure 4.3 shows on the basis of the `read` operation how the configuration policy object in charge has the opportunity to control both the used template and the found entry by applying its `preReadCheck` and `postReadCheck` methods. The hollow arrowheads indicate thereby returned values of the preceding methods.

The agent calls the read operation with the template entry `e1`, which is passed to the policy's `preReadCheck` method. A validated copy, `e2` is used to look up a matching entry `e3`, which again is checked by the `postReadCheck` method. The validated copy `e4` is then passed to the calling agent.

4.3.2 The Read Contract

The collaboration diagram in figure 4.4 shows the detailed steps performed for a successful `read` operation. The consecutive interactions of the participants are the following:

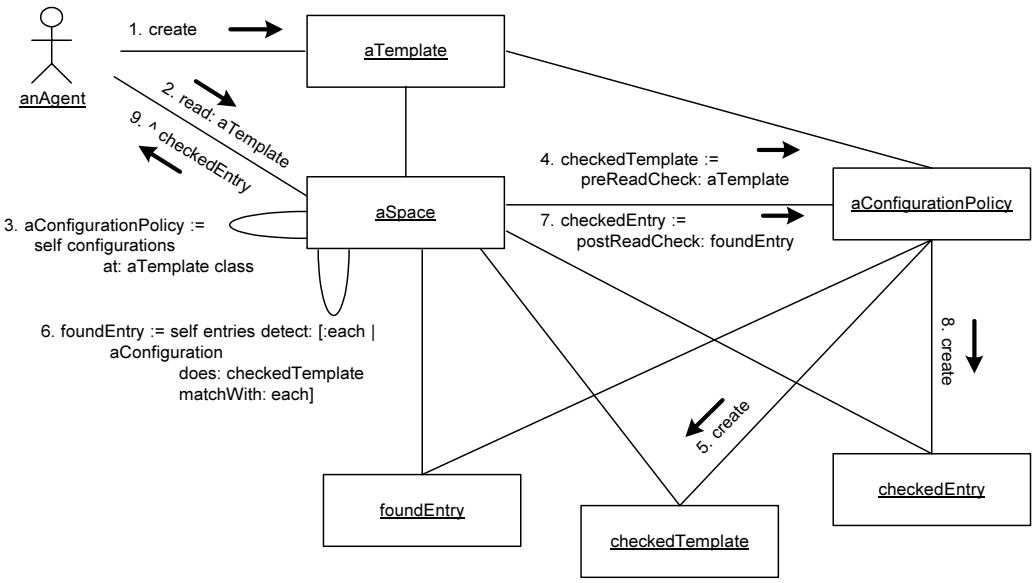

Figure 4.4: The detailed steps for a `read` operation.

1. An agent creates a *template entry* representing the kind of information he wants to retrieve from the space.
2. The agent calls the space's `read` method providing the template as a parameter.
3. The space looks up the configuration policy associated with the template's class.
4. The configuration policy's `preReadCheck` method is called with the template, and the configuration policy returns a *checked template*. This is a validated copy of the original template which may be modified, depending on the implementation of the check method and on the actual values in the template.
5. The space iterates over its entries collection asking the policy for each if it matches with the checked template.
6. The first *found entry* that matches is passes to the configuration policy's `postReadCheck` method which again returns a possibly modified copy of it as *checked entry*.
7. The space returns the validated entry to the agent.

All `read` variants (also `blockingRead`, `readAll`) apply the same pre- and post-hooks before and after the retrieving, the same holds for the variants of `take`.

4.3.3 The Take Contract

The take contract is analogous to the one for read accesses. The difference is that the called hook methods are `preTakeCheck` and `postTakeCheck` and a successful lookup results in the removal of the matching entry from the space.

4.3.4 The Write Contract

The write contract again differs in using its own hook methods. Additionally, the `write` operation has to check if any pending blocking operation waits for an entry like the one just written. Therefore the templates of the blocked accesses must be scanned for a match with the new entry. Readers can all be resumed to get a copy, of the takers only the first will be served (in FIFO order). The same holds for the subscriptions for notification (see section on event notification, 6.5). If the new entry matches any of the subscribers' templates, they have to be notified by issuing the corresponding event.

4.3.5 Benefits

With the described contracts, the hook methods allows a user to specify many useful variants of a space's behavior. This offers advantages like:

- Protection: The space can ensure that the entry put in the space holds certain properties and the space agent is ensured that the retrieved entry is coherent.
- Shifting Responsibility: Instead of requiring the space agents to be responsible for consistency of the space, it is the space itself that does so by invoking the configuration policy that controls its state.
- Simpler Space Protocols: The design of the necessary protocols becomes easier to develop and maintain, the applications end up more robust, and cause less network traffic.
- Hiding Space Administration: The space can perform administrative tasks that are hidden from the space agents.
- Basically *any desired action* may be triggered.

4.3.6 Caveats.

The triggered actions in all of these hooks methods of the configuration policies may easily cause loops. When, for instance, a `preReadCheck` method of a configuration policy calls the `read` method with an entry of the same class as the policy is registered for, there will be an infinite loop, if no precaution is taken. See the garbage collection case study in 5.5 for an example how to work around this problem, to allow the policy to access the space with the same entries it is registered with.

4.4 Implementation

After presenting the design of the framework, we proceed with a short discussion of the most important aspects of its realization, namely with the `OpenSpace` methods handling the registration of the policies, their integration, and the synchronization aspects.

`OpenSpace` is a subclass of `NamingClient`, a helper class of our framework, which implements the common code for using the `SpaceServer`'s naming service. The spaces are registered there under their `spaceName`. They hold collections with their `entries` and their `configurations`. A mutex variable, an instance of the Smalltalk class `RecursionLock`, is used to protect all space operations from concurrent access. Additionally the spaces hold reservation lists to collect pending blocking accesses and an event manager for the notification handling.

4.4.1 Registering Configuration Policies

In order to use an entry with the space, its class must first be registered with a corresponding configuration policy. The class `OpenSpace` offers the method `registerEntryClass`, which is called via the space server. Figure 4.5 shows its implementation. If the given entry class is already registered an exception is raised, if not, a new instance of the specified policy class is created and is stored in the `configurations` dictionary with the entry class as key. On the last line the update hook of the policy is called to perform any transitional actions for the new registration (see 5.5.1 for an example of its use). The method's counterpart, `unregisterEntryClass`, just removes the binding from the spaces dictionary `configurations`. Note that the whole operation is protected with the mutex variable.

```
NamingClient subclass: #OpenSpace
 instanceVariableNames: 'spaceName entries configurations mutex
        readReservations takeReservations eventManager requests'

OpenSpace >> registerEntryClass: anEntryClassSymbol
             withConfigurationPolicy: aPolicyClassSymbol
 "Add an association of the entry class with the according
 policies to the receiver's policies dictionary."
 | entryClass policy |
 mutex critical:
   [entryClass := Smalltalk at: anEntryClassSymbol.
   "Check first if the entry class is not yet registered"
   (configurations includesKey: entryClass)
     ifTrue:
       [self error: entryClass printString,
                    ' is already registered!'].
   policy := (Smalltalk at: aPolicyClassSymbol) for: self.
   configurations at: entryClass put: policy.
   policy updateOldEntriesOfClass: entryClass]
```

Figure 4.5: The registration of an entry type with a configuration policy.

4.4.2 Retrieving Operations

Once registered the policies have to be called from each of the space operations, before and after their actual access. Each operation first has to lookup the corresponding policy object, figure 4.6 shows the according helper method. The space

```
OpenSpace >> getPolicyFor: anEntry
 "Answer the ConfigurationPolicy which is associated with
 anEntry's class"
 ^configurations
     at: anEntry class
     ifAbsent: [nil]

OpenSpace >> findEntryMatching: aTemplate underPolicy: aPolicy
 "Find any Entry matching with aTemplate under the given policy"
 ^self entries
          detect: [:each |
             aPolicy
               does: each
               matchWith: aTemplate]
          ifNone: [nil]
```

Figure 4.6: The helper methods used in the `take` operation

access itself is straight forward, for the retrieving the entries collection is asked to `detect` an element matching with the provided policy.

The `take` operation is shown in figure 4.7. If no configuration policy object is found the method returns immediately with `nil`, otherwise the template is validated by the policy. Then the entries collection is scanned to find a matching entry, the result of this scan is passed to the post-take validation. The code of the `read` method is similar, the `matchingEntry` is not removed on success like at the line indicated with `(*)`. Again, the operation is protected from concurrent access.

```
OpenSpace >> take: aTemplate
 "Non blocking try to remove a matching entry from the receiver's
 entries collection. Answer the result of the postTakeCheck"
 | policy checkedTemplate matchingEntry |
 mutex critical:
   [self increaseRequests.
   policy := self getPolicyFor: aTemplate.
   ^policy isNil
      ifTrue: [nil]
      ifFalse:[
        checkedTemplate := policy preTakeCheck: aTemplate.
        matchingEntry := self
                          findEntryMatching: checkedTemplate
                          underPolicy: policy.
        matchingEntry notNil
          ifTrue:
            [self entries remove: matchingEntry.  "(*)"
            self changed: #entriesChanged].
        policy postTakeCheck: matchingEntry]]
```

Figure 4.7: The implementation of the `take` operation in class `OpenSpace.`

4.4.3 Blocking Operations

The blocking retrieval operations suspend the calling agent until a matching entry can be found. Their request is handled with a *thread-per-message gateway* pattern [Lea96], allowing the space to delegate the request to a light weight helper process.

If a `blockingRead` request is not immediately successful, its template is stored in the `readReservations` dictionary as a key. The associated value is a semaphore which is used to suspend a new lightweight process. This process, once resumed, will perform another `read` operation. The process is forked as a

`Promise`, a Smalltalk object that makes a caller of its `value` method wait until it has finished its execution. Since the process is suspended, the promise suspends the calling agent. Figure 4.8 shows the implementation of this mechanism.

```
OpenSpace >> blockingRead: aTemplate
 "Client is blocked until a matching entry can be returned"
 | policy entry semaphore |
 mutex critical:
   [policy := self getPolicyFor: aTemplate.
   ^policy isNil
      ifTrue: [nil]
      ifFalse:
        [entry := self read: aTemplate.
        entry notNil
          ifTrue: [entry] "return immediately if successful"
          ifFalse:
            [semaphore := self reservateForReading: aTemplate.
            [semaphore critical: [self read: aTemplate]]
               promise value]]]

OpenSpace >> reservateForReading: aTemplate
 "Add aTemplate as a key to the readReservations to be checked
 on the arriving of new entries."
 | semaphore |
 ^(self readReservations includesKey: aTemplate)
    ifTrue:[self readReservations at: aTemplate]
    ifFalse:
      [semaphore := Semaphore new.
      self readReservations at: aTemplate put: semaphore]
```

Figure 4.8: The `blockingRead` operation.

If there are multiple `blockingRead` requests issued with equal templates, their blocked processes will all wait for the same semaphore. When a matching entry arrives, they can all be resumed to let their corresponding agents read the entry. If, on the other hand, multiple `blockingTake` requests are waiting for the same entry, only one of them can be served. Therefore the `takeReservations` dictionary holds for each template-key a collection of semaphores which will be signaled in FIFO order.

4.4.4 The `write` Operation

The `write` method differs insofar from the retrieving operations as it potentially has to notify waiting agents. There are three groups that have to be considered:

(1) the blocked readers, (2) the blocked takers, and (3) those agents who had subscribed for a notification (see 6.5 for details). The readers are notified first, giving them a chance to read the arrived entry, before any taker might remove it. The notification subscribers have the lowest priority and are called last. Figure 4.9 shows the helper methods used for the resuming of the blocked calls. They are called by the `write` method in the named order.

```
OpenSpace>>checkReadReservationsFor: anEntry withPolicy: aPolicy
 "Let all agents who have pendent blockingRead's access anEntry,
 if it matches with the template of their request."
  self readReservations keys
    do: [ :template |
      (aPolicy does: anEntry matchWith: template)
        ifTrue:
          [( self readReservations at: template) signal.
          self readReservations removeKey: template]]

OpenSpace>>checkTakeReservationsFor: anEntry withPolicy: aPolicy
 "Let first agent who has a pending take request take anEntry,
 if it matches with the template of its request."
 | template queue |
 template := self takeReservations keys
    detect: [:templ | aPolicy does: anEntry matchWith: templ]
    ifNone: [nil].
 template notNil
    ifTrue:
      [queue := self takeReservations at: template.
      queue isEmpty not
        ifTrue:
          [queue first signal.     "let the first take anEntry"
          queue removeFirst.
          queue isEmpty
            ifTrue:[self takeReservations removeKey: template]]].
```

Figure 4.9: The reservation checks.

The rest of the `write` method is again very similar to `take` and `read`: find the corresponding policy, perform the pre-check validation, add the new entry to the entries collection, do the above-mentioned notification checks, and do the post-check. By default the written entry is returned.

After having presented our framework, we continue with a series of case studies, showing how OPENSPACES can be employed as a coordination medium supporting different degrees of variability.

Chapter 5

Case Studies

To validate OPENSPACES we implemented a simple application and extended it stepwise with features that show the benefits of the properties of our framework. The running example exhibits a classical set of coordination requirements, its extensions and variations satisfy typical real- world needs in the context of distributed and heterogeneous applications.

In the following section we first introduce the application and its standard implementation, then we present the extensions along the axes of OPENSPACES's configurability options.

5.1 An Electronic Market Place

5.1.1 Scenario

Let us consider a typical trading situation: a buyer agent (i.e., someone representing a client) is looking for some goods or services. She wants to inform potential sellers of her needs and publishes her request for instance as an advertisement in a newspaper or at an eCommerce site. The sellers see the request and may react by offering a concrete bid for it. The buyer agent can choose amongst all offers she has found and may accept the one that meets her needs best, as is shown in figure 5.1.

This simple scenario exhibits several classical coordination requirements: the buyer doesn't know the sellers in advance; multiple potential sellers should be informed of the request; neither simultaneous nor synchronous communication between the buyer and sellers is needed; instead, the request should be *published* in some suitable medium; the request can be withdrawn once it is fulfilled (or expired); multiple buyers may wish to publish requests in thematically related media.

This negotiation protocol therefore perfectly matches the characteristics of a blackboard-style architecture as shown in figure 5.2. Every step can be performed by posting a corresponding entry to the blackboard. It can be read or taken (consumed) by the receiver. Participants don't have to know each others' location or name, they just need to know where to find the blackboard.

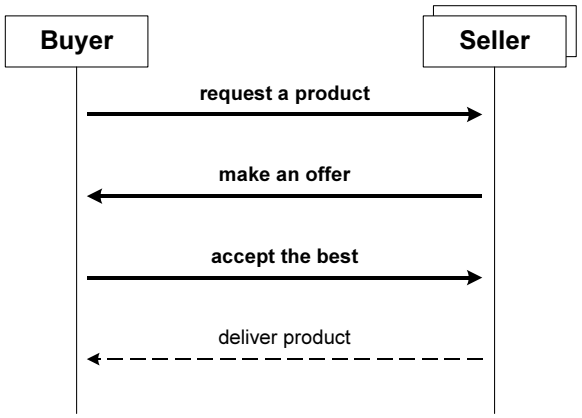

Figure 5.1: The general protocol of a trading negotiation.

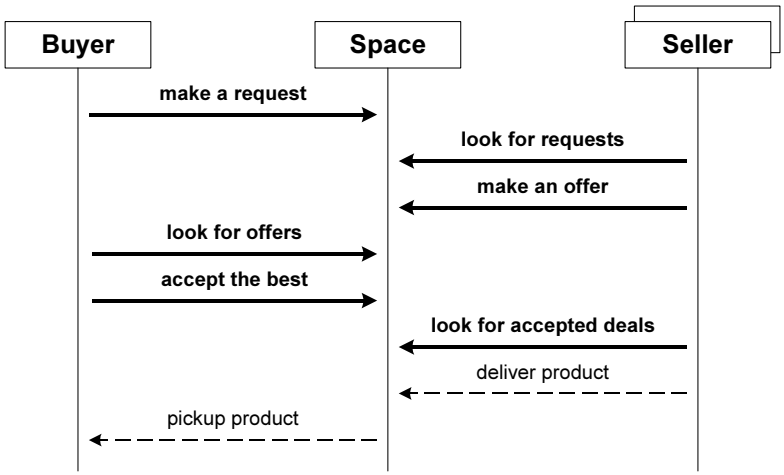

Figure 5.2: The trading protocol adapted to blackboard communication style.

5.1.2 Analysis

Even such a simple scenario as this poses special requirements for the *policies* in place for the blackboard. A minimal implementation would require the following

setup:

- There are two kinds of agents to represent the buyers and the sellers.
- The agents exchange entries (possibly, but not necessarily tuples) representing requests, offers and deals. *Requests* describe the desired product and maximum price. *Offers* describe the offered product and price. A *deal* finally is written to accept a received Offer.
- An offer must reference the request it responds to. A deal must reference the offer it accepts.
- Requests must be readable by anyone interested, but they may only be withdrawn by the issuing buyer, when he is no longer interested in receiving more offers.
- Offers referencing a request may only be read and removed by the initiating buyer.
- Deals may be read and removed by the seller who has issued the referenced offer.
- To divide the market into multiple thematic sections, all entries must have a label with the name of the section they belong to.

5.2 Market Place V.1: Standard Implementation

We now present a standard solution implementing such a market place scenario in a tuple space style. We start with the simplest solution that works and incrementally show how we can specialize and extend the application to provide new features or more refined solutions.

As a concrete entry type to hold the trading data we use *forms* [Lum99]. They contain but a dictionary, which is used to store associations of keys and values. This is a flexible approach since additionally needed values may be added without the need to define new subclasses. We define `Form` as a subclass of `Entry`, its dictionary instance variable is called `bindings`.

The matching algorithm is defined as follows: A template matches a `Form` if (1) the form is an instance of the same class as the template or of a subclass, and (2) the form's bindings contain all the keys of the template's bindings, and (3) their respective values are equal. Additional keys of the form are not considered, i.e. the absence of a label in the template's bindings is interpreted like a wildcard.

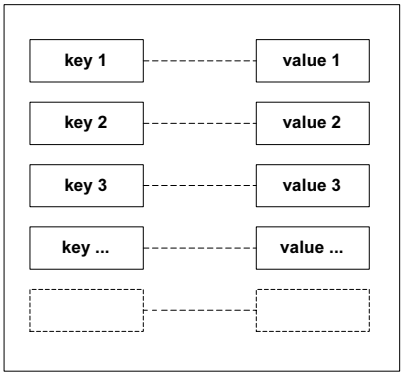

Figure 5.3: The used `Form` entry.

The matching algorithm is defined in the associated configuration policy object. Figure 5.4 shows its implementation in class `FormPolicy` which extends the default `ConfigurationPolicy`.

```
FormPolicy >> does: aForm matchWith: aTemplate
 "Answer true if aForm contains all keys of aTemplate and all
 respective values are equal."
 | ok |
 ok := (aForm isKindOf: aTemplate class)
         and: [aTemplate bindings notNil].
 ok ifTrue:
      [aTemplate bindings keys do: [:key |
        (aForm bindings includesKey: key)
          ifFalse: [ok := false]
          ifTrue:
            [ok := ok and: [(aForm bindings at: key) =
                                (aTemplate bindings at: key)]]]].
 ^ ok
```

Figure 5.4: The matching algorithm for Forms defined in the `FormPolicy` class.

5.2.1 Using the Market Place forms

In the forms' bindings we add the information for the trade communication and the name of the section of the market the form belongs to. As we mentioned in the scenario, a `Form` may be a request, an offer, or a deal. Therefore it gets a binding with the key `#formType` and a value denoting its type (one of `#request`, `#offer`, `#deal`).

A request form can have bindings for a product name, for a description of the product, and for the maximum price the buyer is willing to pay for it. An offer form may describe a concrete product and price, and holds a reference to the request it reacts to. A deal form has a reference to the offer it accepts. See next section for details on the referencing between the forms.

5.2.2 Uniquely Identifiable Entries

To have a uniquely referenceable index for each form, we add a binding for the index and its value to every form when writing the form to the space. To remember the highest index used so far, there is a `TailEntry` to hold its current value. A `MarketAgent` – a specialized `SpaceAgent`, see figure 5.6 – who wishes to append a new form to the market place, takes the tail entry, increments its index value, puts it back to the space and writes the form with the new index to the space (see figure 5.5). Besides remembering the last index used, the `TailEntry` – actually but a counter – acts also as a semaphore, ensuring that only one form is written per index.

```
MarketAgent >> append: aForm
 "Get a new index, add it to aForm and write aForm to the space"
 |template tail newIndex|
 template := (TailEntry onSection:(aForm bindings at:#section)).
 tail := self hostSpace take: template.
 newIndex := tail index + 1.
 tail index: newIndex.
 self hostSpace write: tail.
 aForm bindings at: #index put: newIndex.
 self hostSpace write: aForm
```

Figure 5.5: The MarketAgent's appending method.

A `MarketAgent` that wants to read all requests present in the space, would have to iterate over all indices up to the last one denoted by the tail. As a simplification the agent may use the `readAll` operation provided by OPENSPACES, which is similar to `read`, but returns a collection with *all* entries from the space that match the template.

To start the trading on the market, a `MarketAgent` has to write a new tail entry with the name of the new section and an index set to 0, denoting an empty market section.

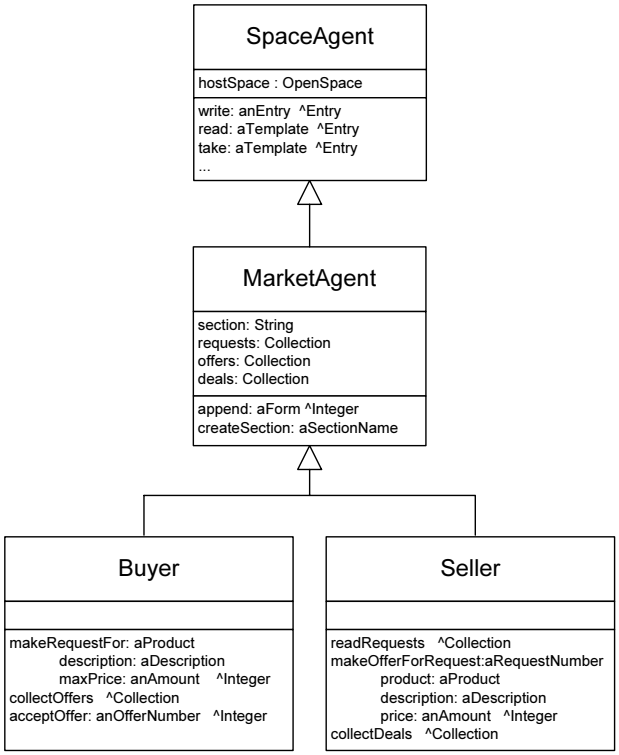

Figure 5.6: Specializing the `SpaceAgent` to participate in the market.

5.2.3 Stepping through a trade

As actual participants we specialize `MarketAgent` to `Buyer` and `Seller`. Their respective protocols support the role specific actions of their respective parts. These are as follows:

- The buyer makes a request by appending a new request form to the market space.
- Sellers get the request when scanning for newly arrived requests.
- Each seller may make an offer for a request by appending an offer form which references the request with its index.
- The buyer scans for offers to her request and takes them from the market.
- When detecting a valuable offer, the buyer accepts it by appending a deal form with a reference to the offer and also the initial request to avoid differences. She then removes the referenced requests.
- By scanning for reactions to her own offers, the seller detects the deal form takes it from the market.

For each participant we have built a simple user interface (see figure 5.7). The `BuyerUI` allows a user to specify a requested product and send a request form to the space. The UI shows the received index for the request. The seller can read the requests and append offers to the market. When the buyer finds enough offers for her requests, she can select the best and send a deal form to accept it. The seller finds the deal and takes it from the market place. All read or removed forms are stored by the participants as references.

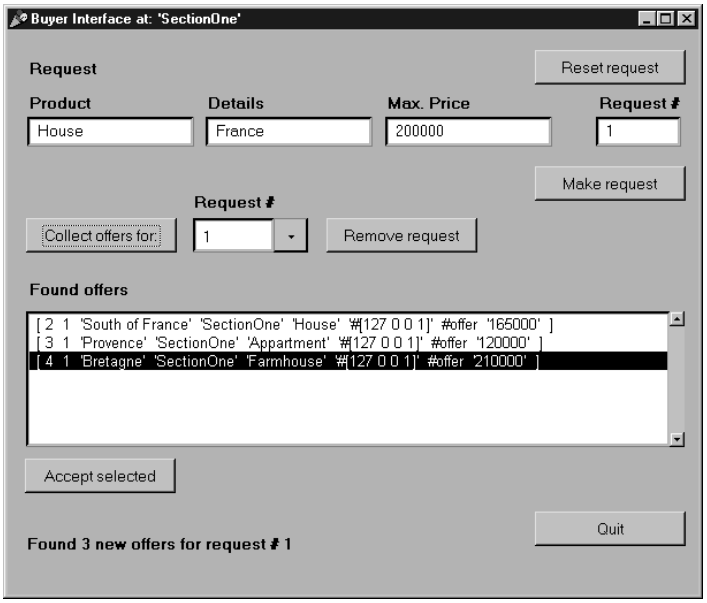

Figure 5.7: The user interface for the buyer.

5.3 Market Place V.2: Consistency Assertions

In the standard version of our Market Place, there is no control over the consistency of the used forms. Their correctness based on *voluntary conventions* to which the agents may adhere or not. Incomplete forms, incorrect indices or referencing would considerably reduce the applications usability.

In order to establish some *consistency assertions*, we defined corresponding configuration policies for each of the three form types. They are applied at each attempt to write a form, rejecting it, if it is not complete and correct.

The checking is made in subclasses of `ConfigurationPolicy`. To become effective, the policies must be registered with the class of entries they will be affecting. We therefore subclassed `Form` creating `Request`, `Offer` and `Deal` (without any changes in the implementation). The corresponding policies, `RequestPolicy`, `OfferPolicy`, and `DealPolicy`, each override the

```
FormPolicy subclass: #RequestPolicy

RequestPolicy >> preWriteCheck: aForm
 "Check aForm for necessary keys. Check if its index is equal
 to the tail's. Check if there is no form with the tail's index
 at the same section."
 |form ok|
 form := super preWriteCheck: aForm.
 ok := ((form notNil
           and: [form bindings includesKey: #section])
           and: [form bindings includesKey: #product])
           and: [form bindings includesKey: #index].
 ok ifTrue:
      [|template tail forms section|
      section := form bindings at: #section.
      template := TailEntry onSection: section.
      tail := self hostSpace read: template.
      ok := ok and: [(form bindings at: #index) = tail index]].
 ok ifTrue:
      [template := Form new.
      template bindings at: #section put: section.
      forms := hostSpace readAll: template.
      forms do: [:each |
         (each bindings at: #index) = (form bindings at: #index)
              ifTrue: [ok := false]]].
 ^ok
    ifTrue: [form]
    ifFalse: [nil]
```

Figure 5.8: A consistency asserting policy for request forms.

`preWriteCheck` method to specify the necessary checks on the forms before they are written to the space. The forms are accepted only if they are correct, otherwise they are rejected by returning `nil`.

The `RequestPolicy` checks if the request form is complete and if its index actually is unique and equal to the tail entry's index. Figure 5.8 shows the code of this consistency check. The `OfferPolicy` checks if the offer includes a reference to a request which is currently present at the space. The `DealPolicy` checks if the deal includes references to an offer and a request.

With the same approach we may increase consistency of the system even more by not allowing an agent to write the same form twice. Note that the form and policy classes can be used with the new policies as soon as they are defined and registered at the space.

5.4 Market Place V.3: Automatic Index Handling

The index incrementing procedure for writing market forms is somewhat awkward. It leads to boiler plate code in the client, and is prone to errors. We want to reduce the responsibility of the market agents - and also the network traffic - by doing this at the space.

We specialize the configuration policies to handle the indices automatically. The agents append forms without adding an index binding. The policy takes care of the tail business and sets the index of the form accordingly. Since the write operation of the space returns entry that has actually been written, the agent checks this for the received index.

```
AutomaticIndexPolicy >> preWriteCheck: aForm
 "Get current tail index by taking the tail entry, incrementing
 its index and putting it back. Enter the new index to aForm."
 |checkedForm|
 checkedForm := ((aForm notNil
                    and: [aForm bindings notNil])
                    and: [aForm bindings includesKey: #section])
                      ifTrue:[aForm]
                      ifFalse:[nil].
 checkedForm notNil
   ifTrue:
     [| template tail newIndex section |
     section := aForm bindings at: #section.
     template := TailEntry onSection: section.
     tail := self hostSpace take: template.
     newIndex := tail index + 1.
     tail index: newIndex.
     self hostSpace write: tail.
     aForm bindings at: #index put: newIndex.
     self hostSpace write: aForm.
     checkedForm bindings at: #index put: newIndex].
 ^checkedForm
```

Figure 5.9: The automatic index handling at the configuration policy

In Figure 5.9 the `preWriteCheck` method passes the validated form back to the space's writing operation where it will be inserted into the space instead of the original form. Since the configuration policy is local to the space, the space accesses are done quickly and cause no network traffic.

This policy allows us to reduce the responsibilities the agent has to fulfill, allowing it to be 'thinner'. Altogether this modification reduces the risk of errors, improves consistency of the market, and thereby assures better performance.

5.5 Market Place V.4: Garbage Collection

Automated handling of outdated forms is a desirable requirement. Without such support, obsolete requests and forgotten offers can easily start to clutter the space. One possible solution for this problem is to add a time stamp to every entry being written to the space. After the expiry of the lifetime of the entry, it will be discarded. This may be after a standard duration defined for all entries or after an individual amount of time that is specified in each entry's fields.

We implemented a Market Place variant with a default expiration time for all forms. A special configuration policy – the `LeasingPolicy` – is in charge of handling the garbage collection. Figure 5.10 shows the specification of its hooks methods. The `preWriteCheck` method add the time stamp, and the pre–retrieving hooks trigger the deletion of expired forms. This is sufficient for consistency since every access ,,sees" a freshly updated view of the Space.

```
FormPolicy subclass: #LeasingPolicy
 instance variables: 'defaultLeasingTime calling'.

LeasingPolicy >> preWriteCheck: aForm
 "Add the current time to the form to be checked for expiration."
 | checkedForm |
 checkedForm := super preWriteCheck: aForm.
 checkedForm notNil
    ifTrue: [checkedForm bindings
                           at: #arrivingTime
                           put: Time now printString].
 ^checkedForm

LeasingPolicy >> preReadCheck: aTemplate
 "If called by a client initiated read operation, perform the
 cleanup, if called by the receiver, just return aTemplate."
 | checkedTemplate |
 ^self calling
    ifFalse:
       [checkedTemplate := super preReadCheck: aTemplate.
       self cleanupSectionOfTemplate: checkedTemplate]
    ifTrue:
       [aTemplate]

LeasingPolicy >> preTakeCheck: a Template
  "Analogous to preReadCheck: "
```

Figure 5.10: The policy for automatic garbage collection.

The method `cleanupSectionOfTemplate` initiates the removal of the

outdated forms in the space. The removal is done with the sequence: `readAll` forms at the same market section, scan through the found collection, and `take` each outdated form from the space. Since the necessary space operations use forms, they are subject to the ruling of the policy object itself. To prevent the policy from endless loops from calling itself, the method `cleanupSectionOfTemplate` enters a „cleaning-mode" during the actual cleanup. The instance variable `calling` holds a boolean value indicating the state of the execution: it is set to true when the policy itself will perform space operations with forms, the `preReadCheck` and `preTakeCheck` hooks will bypass the internal cleanup in this case. The hook methods of the policy are called during the space operations, which are synchronized with the `mutex` variable (see the section on the implementation 4.4). The temporary bypassing of the configuration policy therefore will not enable uncontrolled access.

```
LeasingPolicy >> calling
 calling isNil
   ifTrue: [calling := false]
 ^calling

LeasingPolicy >> cleanupSectionOfTemplate: aTemplate
 "Check if aTemplate is valid. Enter 'cleaning-mode', perform
 cleanup, exit 'cleaning-mode'. Answer aTemplate if successful,
 else nil."
 aTemplate notNil
   ifTrue: [| section |
     self calling: true.
     section := aTemplate bindings at: #section.
     self throwAwayOutdatedFormsAtSection: section.
     self calling: false.
 ^aTemplate

LeasingPolicy >> throwAwayOutdatedFormsAtSection: aName
 | template forms arrived |
 template := Form with: (Array with: #section with: aName).
 forms := hostSpace readAll: template.
 forms do: [:each |
   (each bindings includesKey: #arrivingTime)
     ifTrue:
       [arrived := each bindings at: #arrivingTime.
       (arrived addTime: defaultLeasingTime) < Time now)
         ifFalse: [hostSpace take: each]]       "re-
move if too old"
```

Figure 5.11: The internals of the LeasingPolicy.

5.5.1 Reconfiguration

As soon as a new configuration policy class is defined and is associated with some entry classes at a space, it will be used, i.e. the next space operation with an affected entry will be ruled by the policy's matching algorithm and access checking methods. Thus the space can be dynamically reconfigured.

In our example with the `LeasingPolicy`, we must, however, adapt the forms that are already in the space. When establishing this policy, we want all forms to be subject of its control. We use the hook method `updateOldEntriesOfClass` to add the current time as arrival time to these forms. Figure 5.12 shows the corresponding code.

```
LeasingPolicy >> updateOldEntriesOfClass: anEntryClass
 "Take all present forms from the space and rewrite them. Like
 that they are supplied by the policy with a binding denoting
 they had just arrived."
 | template forms |
 template := anEntryClass new.
 self calling: true.                          "prevent loops"
 forms := hostSpace takeAll: template.
 forms
   do: [:each | hostSpace write: each].
 self calling: false
```

Figure 5.12: The update method for the dynamic activation of the LeasingPolicy.

The method removes all forms from the space and writes them back again. Since the update hook is called after the activation of the policy, this triggers the new `preWriteCheck` hook method, which adds the arriving time binding to the form. Again, to avoid an infinite loop, we use the construct with the state variable `calling`.

5.6 Monitoring Space Accesses

After the variations on the Market Place we are now presenting two small examples using the configurability of OPENSPACES in a more general context. Both are concerned with security issues. The first shows how the configuration policies can be used for monitoring all accesses at the space by reporting them to a log file. This may be used for purposes of analysis or surveillance.

Figure 5.13 shows the code that implements the logging. The pre- and post-hook methods simply report each attempt to perform a space operation or success-

fully completed ones, respectively. The affected forms are included, too. Together with the current time, a string describing each event is appended to the log file.

```
AutomaticIndexPolicy subclass: #LoggingPolicy
 instanceVariableNames: 'logPath '

LoggingPolicy >> preReadCheck: aTemplate
 "Perform default super check. Report attempt to read with the
 provided template and the checked one to the log."
 | checkedTemplate |
 checkedTemplate := super preReadCheck: aTemplate.
 self
   report: 'Attempt to read'
   using: aTemplate
   resultingWith: checkedTemplate.
 ^checkedTemplate

"Other hook methods analogous"

LoggingPolicy >> report: aString
                 using: anEntry
                 resultingWith: aCheckedEntry
 self appendToLog:
        Time now printString, ': ', aString,
        ' using: ', anEntry printString,
        ' resultingWith: ', aCheckedEntry printString.

LoggingPolicy >> appendToLog: aString
  | file |
  ^self logPath asFilename canBeWritten
    ifFalse:[self error: 'Cannot write to the log file!'.]
    ifTrue:[
      [file := self logPath asFilename readAppendStream text.
      file
        setToEnd;
        nextPutAll: aString;
        cr;
        flush.
      ] ensure: [file close]]
```

Figure 5.13: A Logging Policy reporting each space access to a log file.

The example provides a detailed log for analyzing the space communications. With a similar policy we can implement a *restoring option*, using the entries' `storeString` method which answers a string containing the code to create an identical object. By logging these store-strings the state of the space's entries collection can be restored if necessary.

5.7 Protecting Single Entries

For confidential information it would be desirable to have a mechanism preventing unauthorized agents from accessing it. As an example of a security protocol that can be realized with OPENSPACES we implemented a policy which allows single forms to be locked with secret keys. This mechanism is inspired by [BOV99].

Each form that is written to the space can be protected by adding a binding with the label `#secretKey`. The matching algorithm tests the presence of this key for each form it checks. If the form is protected, the template must present such a binding, too, and their values must match. We can employ two different locking strategies, symmetric or asymmetric. Two *symmetric keys* match, if they

```
FormPolicy subclass: #LockingPolicy

LockingPolicy >> does: aForm matchWith: aTemplate
 "If aForm matches according to standard form matching (super),
 check if it is locked with a key. If so, aTemplate must present
 a key that matches with aForm's key."
 | templateKey formKey |
 formKey := aForm bindings at: #secretKey ifAbsent:[nil].
 templateKey := aTemplate bindings at:#secretKey ifAbsent:[nil].
 ^(super does: aForm matchWith: aTemplate)
    and: [formKey isNil
         or: [self doesKey: templateKey matchWith: formKey]].

LockingPolicy >> doesKey: templateKey matchWith: formKey
 "Here we can implement some symmetric or asymmetric strategy
 for the locking. By default test for equality."
 ^ templateKey = formKey

LockingPolicy >> postTakeCheck: aFoundForm
 "If aFoundForm has been protected with a secret key, remove it
 to keep it secret"
 aFoundForm bindings removeKey: #secretKey ifAbsent: [].
 ^aFoundForm

LockingPolicy >> postReadCheck: aFoundForm
 "If aFoundForm has been protected with a secret key, remove it
 from the clone to keep it secret"
 | clone |
 clone := aFoundForm getClone.       "deep copy"
 clone bindings removeKey: #secretKey ifAbsent: [].
 ^clone
```

Figure 5.14: An implementation of a security protocol with locks for the forms.

are equal, two *asymmetric keys* must complement each other according to some algorithm. Figure 5.14 shows the conceptual implementation using symmetric keys.

The important implication of this policy is, that locked entries are completely invisible to agents not presenting the correct key. Moreover, when using a asymmetric locking strategy, the used keys to lock the forms should be kept secret to prevent misuse. To this end, the post-retrieving hook of the policy removes them before passing the forms to the agents.

5.8 Characteristics of the Examples

The presented case studies expose a set of possibilities to modify the behavior of a data space provided by OPENSPACES. The coordination medium can be charged of several tasks to perform, triggered by attempts to access it and by successful access operations. The following possibilities are provided:

- *Rejecting Entries.* The use of incorrect, incomplete, or redundant entries may be prevented.
- *Modifying Entries.* Entries may be completed or corrected before they are written to the space or used as templates.
- *Reading Access to the Space.* The configuration policy objects may themselves access the space. Using the `read` operation they can control its state.
- *Modifying Access to the Space.* Using the `write` or `take` operations they can modify the state of the space. This can be useful for many purposes, like shown in the examples with automated indices or the garbage collection.
- *Performing Helper Tasks.* The policy can be charged of small tasks for optimizing the space protocol of an application.
- *Updating Present Entries.* For a smooth transition when changing the active policies , the entries already at the space can be adapted to the needs of the new policy.
- *External Side Effects.* If necessary, any action may be triggered by accesses to the coordination medium.

These options may be used to realize many improvements compared to the standard tuple space communication, including consistency, security, privacy, less network traffic, improved efficiency, reduced proneness to errors, and more.

Table 5.1 shows an overview on the realization of the mentioned features within our case studies. The standard implementation, of course, does not use any special configuration. For completeness the last two columns show the two examples that are presented in the next chapter: the notified agents are set up on the standard configuration, the Java client participates in the Market Place V.3 setup with automatic indices.

Actions at Policy	Example							
	Standard Implementation	Consistency Assertions	Automatic Indices	Garbage Collection	Monitoring Accesses	Locking Entries	Notified Agents	Java Client
Rejecting Entries	-	√	-	-	-	√	-	-
Modifying Entries	-	-	√	-	-	√	-	√
Reading Access to the Space	-	√	√	√	-	-	-	√
Modifying Access to the Space	-	-	√	√	-	-	-	√
Performing Helper Tasks	-	-	√	√	√	√	√	√
Updating Present Entries	-	-	-	√	-	-	-	-
External Side Effects	-	-	-	-	√	-	-	-
Use of Notification Service	-	-	-	-	-	-	√	-

Table 5.1: Overview on the configuration tasks in the presented examples.

After studying the example applications, we continue in the next chapter with a discussion of the distribution aspects of the framework.

Chapter 6

Distribution and Heterogeneity

OPENSPACES aims to provide a coordination model for distributed applications, therefore it features remote access to the data space abstractions. Clients may reside anywhere on the Internet, communication between the spaces and their clients is realized with CORBA on top of TCP/IP.

The requirements concerning heterogeneity are (1) to be usable in an environment with multiple operating systems, and (2) to provide access for agents written in several programming languages. OPENSPACES fulfills the first requirement for both the spaces on the server side and the standard client agents. The framework is written in Cincom Visual Works Smalltalk, which runs under four major platforms: Solaris, Windows, Linux, Macintosh. The code is fully portable, thus an `OpenSpace` may work on any of them. The same holds for the included client class `SpaceAgent`.

The second heterogeneity requirement is fulfilled by using CORBA as the communication layer between a space and its clients [OMG98]. Since CORBA is a standard supported by a large number of vendors, there are implementations on virtually every operating system/programming language combination. This establishes a good basis for building a heterogeneous coordination platform.

In this chapter, we first discuss general issues of distributed applications and CORBA, then we present the design of the CORBA-layer of OPENSPACES. We introduce a Java client agent participating in the example application and discuss the experiences made with the implementation. Finally a variant of the Market Place using distributed event notification is presented.

6.1 Distributed Applications

With today's ubiquitous networks more and more applications are built using distributed architectures. This offers several advantages like:

- Powerful servers can be used by many clients.
- Remote access to shared information is possible, e.g. central databases.
- Maintaining the consistency of a system becomes easier.
- Applications can be split: distributed components collaborate, each specialized to a different tasks. There is no need anymore to do everything everywhere.
- Thin clients become possible. They are less demanding with respect to hardware, and also become easier maintainable.
- A distributed application can sum up the power of the participating machines („virtual parallel computing").
- Support for mobility of users or software agents.

The most common architecture for distributed applications is probably the Client/Server setup, e.g. several clients using a central database server. An architecture with a finer grained division of tasks is called *N-tier*, comprising for example a Business Server, a Web Server, and many Browser clients. For a detailed discussion of distribution issues, see for example [CDK94].

6.2 Introduction to CORBA

The object-oriented programming paradigm is particularly suited for distribution: objects encapsulate data and corresponding methods, access to the object's data and services is defined by their (public) interface. The internals of the implementations are private to the objects: as long as the interface contracts are fulfilled, changes can be applied without affecting the collaboration with other objects .

CORBA [1] offers a mechanism to do exactly the same with remote objects. The interfaces are „published" by declaring them in an IDL module. Each participating object uses a local ORB (*Object Request Broker*) which forwards the requests and answers between the calling object and the remote ORB.

As shown in figure 6.1 a request of a client object is forwarded by its local ORB to the remote ORB of the server object. The return values are passed the same way in the reverse direction. The client and server roles thereby can change for each interaction.

The translation of complex objects into data streams, that can be sent over the network, is called *marshalling*, the reverse process *unmarshalling*. The ORB is

[1] CORBA (*Common Object Request Broker Architecture*) is a standard issued by the Object Management Group (OMG), a vast international organization whose members are an (almost) complete who-is-who in the area of information technology. For more information on CORBA refer to the specification by OMG [OMG98], or to introductory books like [Mon99].

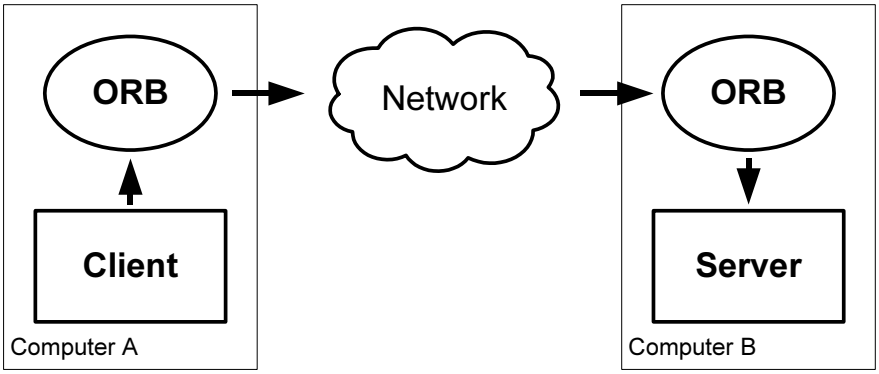

Figure 6.1: The propagation of a remote request via ORBs.

responsible for this conversion. The data types of each object that will be marshalled, must be declared in a corresponding IDL module. The modules are registered in the *repository* where the ORB can access them.

In order to call a method on a remote object, the caller object needs a reference to it. There are two ways of retrieving one: using a *Interoperable Object Reference* (IOR) or via the CORBA *Naming Service*. An IOR is a string-encoded form of an object reference containing information like IP address and port number of the object's ORB, its CORBA name, and a unique identifier. The decoding of an IOR results in a *remote object reference* that is used as a proxy [GHJV95]. The Naming Service is a CORBA service that offers lookup of an object by its CORBA name. This name can be structured hierarchically in several naming contexts to provide uniqueness.

The CORBA Services (COS) offer solutions to typical problems in the context of distributed computing. In addition to the mentioned Naming Service there are Event Notification (see 6.5), Transaction and Concurrency support, Life Cycle Service, and more. Most implementations provide only a subset of all services proposed by the OMG.

6.3 The CORBA–Layer of OPENSPACES

After these general aspects of distribution and CORBA, we continue with the concrete realization of the distribution mechanisms in OPENSPACES.

6.3.1 Basics

OPENSPACES uses Cincom Distributed Smalltalk (DST 5.7) which is CORBA 2.0 compliant. DST provides five important CORBA Services: Naming, Lifecycle, Events, Concurrency and Transactions. OPENSPACES uses the Naming Service for enabling an agent to connect to a space and the Event Service for the notification mechanism (see section 6.5).

Since OPENSPACES runs on top of CORBA, any external client may use an installation of OPENSPACES. The only prerequisites are the use of an ORB and the implementation of the necessary data structures for the entry types of the application. There is no other restriction concerning language or platform for a client implementation.

Agents need references to their „host space" (or to several spaces, depending on the application). Two ways to find a selected space are offered: via the an IOR or via the Naming Service which holds a reference to the Space Server (see 4.2.5). The spaces offer the access operations as public interface, declared in the OPENSPACES IDL module. The different entry types of an application must be declared in additional modules which must be known to both the space and the agents.

6.3.2 IDL Definitions for OpenSpaces

The IDL declarations for OPENSPACES include the interface to the spaces, the entry data types, and some base types, used for the entries and the parameters.

`OpenSpaceInterface` is the CORBA interface for the space class `OpenSpace`. With this declaration a local ORB will provide an agent with a *remote object reference* – a kind of a proxy – that can be used to invoke the operations of a remote space. Figure 6.2 shows the interface declaration.

The `#pragma selector` direction enforces (and documents) the mapping between a method defined in the IDL and its corresponding Smalltalk selector. The line `#pragma selector write write:` binds the IDL method `write` (to be defined thereafter) to the Smalltalk `write:` method. Pragmas are implementation specific extensions, DST also uses them to bind Smalltalk classes to IDL typedefs (see figure 6.3). If another implementation does not understand a pragma, it simply ignores it.

The entry arguments and return values of the space operations are declared as of type `ANY`, a pseudo type that stands for any *known* type. This construct keeps the declaration open to be used with new entry types defined by an application. This is necessary because the entries have to be declared as `structs`, which do not support inheritance. In CORBA `structs` are the only way to pass objects by value and not as remote reference. Since entries are to be put into the space without dependencies on remote objects, we need to transfer actual copies

```
interface OpenSpaceInterface {
    #pragma selector write write:
        any write( in any anEntry );
    #pragma selector read read:
        any read( in any aTemplate );
    #pragma selector take take:
        any take( in any aTemplate );

    #pragma selector blockingRead blockingRead:
        any blockingRead( in any aTemplate );
    #pragma selector blockingTake blockingTake:
        any blockingTake( in any aTemplate );

    #pragma selector readAll readAll:
        OrderedCollection readAll( in any aTemplate );
    #pragma selector takeAll takeAll:
        OrderedCollection takeAll( in any aTemplate );
};
```

Figure 6.2: Declaration of the OPENSPACES-interface.

of them.

The keyword `in` declares a parameter as one–way, it is only passed from the sender to the receiver of the call. `out` would stand for an argument only used as return value. An `inout` parameter is passed in both directions: the client can pass a value to the server, which can modify it, similar to a Pascal `var` parameter. However, we only use the `in` variant, the other modes are not really necessary for clean object-oriented programming. In Smalltalk, it is the standard way of passing a parameter: arguments may not be reassigned within a method.

For the bulk–retrieving operations we defined `OrderedCollection` as sequences of 0...n elements of type `ANY`. This is the closest CORBA type for the Smalltalk `OrderedCollection` that are returned by the operations. Figure 6.3 shows the IDL declarations for a set of general base types and the entries used in our examples. The base types are actually predefined in DST, the overriding serves for documentation purpose. For practical reasons we restricted `Association` to string type values (see 6.4.2 for explanation).

6.3.3 Impact on the Design of Applications

CORBA provides for heterogeneous access to the spaces. It implies, however, some restrictions for the design of applications using OPENSPACES.

```
#pragma class OrderedCollection OrderedCollection
    typedef sequence<any> OrderedCollection;
#pragma class Association Association
    struct Association {
       any key;
       string value; };
#pragma class Dictionary Dictionary
    typedef sequence<Association> Dictionary;
#pragma class Symbol Symbol
    typedef string Symbol;
#pragma class ByteString ByteString
    typedef sequence<char> ByteString;

#pragma class Form Form
    struct Form {
         Dictionary bindings;
    };
#pragma class TailEntry TailEntry
    struct TailEntry {
     long index;
     string section;
};
```

Figure 6.3: Declarations of base types and of the used `Form` entry.

Typing: Every argument and return value of remote method invocations must be of a type declared in the IDL modules. This is a small overhead when using a language with strong typing. With a dynamically typed language like Smalltalk this requires some caution when designing the agents for the space application. Common habits like methods returning errors or strings instead of the usual types will produce runtime errors like the `BAD_PARAMETER` exception.

ANY type: The `ANY` construct could basically be a flexible alternative to the strict compulsion for typing. The handling of `ANY` values however, causes a lot of programming overhead when converting from and to concrete values (see also section 6.4.2). Moreover, the use of `ANY` is less efficient.

NIL wildcards: For entry types with multiple fields containing single values, a typical approach to specify a wildcard, is to set a field's value to `nil`. This works fine in a local setup without distribution. CORBA, however, offers no means to transport a void value. There is a limited workaround for that. A `union` type can be declared in the IDL, containing a field with either some value of a known type, or a remote object reference. When marshalling a `nil` value, a reference on the void object is generated. Experiences with the implementation of the Java client

however, showed that this works only within Smalltalk. The Java ORB generated a marshalling error for all possible variants. The difference seems to origin in the Smalltalk treatment of *anything* as an object, even the `UndefinedObject` („`nil` "). In Java, a `null` value is not an object and there is no way to generate a reference to it, thus it can not be used as wildcard. A application must therefore define its protocol with one of the following options:

- Definition of a `RemotableNil` class (e.g. for the Java part) to realize the same workaround as is possible in Smalltalk. A wildcard in a template is then defined with an instance of `RemotableNil`, the corresponding `union` definition in the IDL will cause it to be processed as a remote reference, as described above.The corresponding matching algorithm recognizes this reference as a wildcard. For the case where a reference to an existing object should be transported in an entry, the IOR of the referenced object could be used.
- A simple but not very flexible workaround is to use a special value to denote the wildcard, e.g. -1 for natural numbers. However, if such a protocol is extended, it is likely that collisions have to be explicitly avoided.
- The `Form` entries of our case study in chapter 5 do not need any special wildcards. The presence of a key–value binding is interpreted as an actual value, the absence as a wildcard. This approach is very flexible.

NIL returns: The `nil` returns of failed space operations cause the same effect to occur. If an operation returns an `UndefinedObject`, it will be processed according to the default definition for `Object`, which is an interface. Therefore the client will receive a remote object reference as a result. Since entries are transported *by value*, the fact that a reference is sent can be interpreted as a failure, the same way as the `nil` in the local setup is interpreted.

An alternative would be what seems to be a common practice in CORBA programming. A special `FailureException` is defined, which will be thrown instead of returning `nil`. This seems, however, to be a sort of misuse of the concept of exceptions. Instead of signaling an exceptional state of the program execution, the exceptions would indicate an ordinary return value of an operation.

6.4 The Java Client

To test OPENSPACES as a coordination medium in a heterogeneous setup, we implemented a client agent with the Java programming language. This agent participates as a buyer in the previously presented Market Place example. Its design is analogous to the one of the Smalltalk participants (see figure 5.6): a

general `SpaceAgent` class encapsulates the handling of the standard space operations, the `MarketAgent` implements the necessary protocol for using the Market Place, and `Buyer` implements the specialized buyer role.

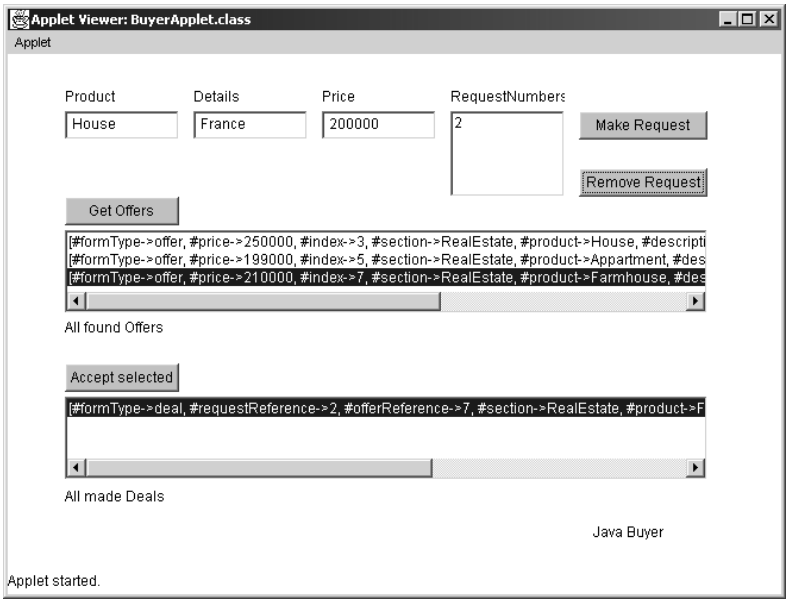

Figure 6.4: The GUI of the Java Buyer.

6.4.1 IDL–to–Java

Language Mappings. The OMG standard defines several mappings between IDL declarations and the corresponding constructs in the host languages. These mappings include definitions of language-specific data types and interfaces to access the ORB. They guarantee a certain degree of portability between different ORB implementations (see [OMG98] p. 2.8). They also provide the possibility to automatically generate skeleton code to adapt the IDL definitions within a programming language.

The `idl2java` Tool. For the implementation of our Java client we used the CORBA ORB of Inprise Visibroker 3.0 together with the Java jdk 1.1. Visibroker offers an IDL compiler tool called `idl2java.exe`. It is used to generate Java source files from IDL definitions conforming to the standard OMG IDL–Java mapping. For an introduction to Visibroker, see for instance [PWGB99].

Basic types like `boolean`, `char`, and `string` are directly mapped to their Java equivalent. There are three *constructed types* that originate from the pro-

gramming language C : `Enum`, `Struct`, and `Union`. They all are converted to final public classes with the necessary attributes and a set of constructor and helper methods. Each `typedef` is either mapped to its original (primitive) Java type or to a generated Java final class with the corresponding attributes. Figure 6.5 shows an extract of the generated Java `Association` class for the corresponding IDL `struct`.

```
struct Association {                          // IDL declaration
  any key;
  string value;
};

final public class Association {              // Generated Java class
  public org.omg.CORBA.Any key;
  public java.lang.String value;

  public Association() {}

  public Association(org.omg.CORBA.Any aKey,
                     java.lang.String aValue) { ... }
  // ...
}
```

Figure 6.5: The generated Java class for the IDL definition codeAssociation.

Additionally to each generated class `Xy`, a `XyHolder` and a `XyHelper` class is generated. The holder class is a wrapper that can be used to supply an object as a `out` or `inout` parameter. Like Smalltalk, Java does not support assigning a new value to a method parameter. If the holder is sent, its content can be exchanged. The helper class provides utility methods for operating on the instances of the corresponding class (`Xy`). This mainly serves to prevent the bloating of the mapped classes. The utilities include conversions from and to `any` types (see also 6.4.2), access to type and ID information. Figure 6.6 shows a part of these two classes supporting the `Association` class.

Entries as Structs. Since entries have to be encoded as structs (see 6.3.2), their hierarchical structure cannot be represented in the IDL. Structs are sufficient since the entries do not need any behavior. For semantic clarity and also for some implementation ease we changed the generated classes to form an inheritance structure according to the one used on the Smalltalk side. The abstract class `Entry` forms the root of the hierarchy. There was no problem running the modified code. However, the arguments and return values of the space operations had to stay declared as `ANY`.

```
abstract public class AssociationHelper {

  // inserts the association into the ANY
  public static void insert(org.omg.CORBA.Any any,
                            OpenSpaces.Association association) {
  // ...}

  // extracts an association from the ANY
  public static OpenSpaces.Association
     extract(org.omg.CORBA.Any any) { ... }

  public static org.omg.CORBA.TypeCode type() { ... }

  public static java.lang.String id() {
    return "IDL:OpenSpaces/Association:1.0";
  }
  // ...
}

final public class AssociationHolder implements
                              org.omg.CORBA.portable.Streamable {
  public OpenSpaces.Association value;
  // ...
}
```

Figure 6.6: The generated `Helper` and `Holder` classes for the associations.

6.4.2 ANY overhead

In the Java class `SpaceAgent` the basic space protocol is implemented. Since every parameter has to be of type `ANY`, we added a method `asAny()` to each of the entry classes. They return, with the help of their associated `Helper` classes, their `ANY` equivalent (see figure 6.7). The space agents use this to convert the entries and the templates to correct `ANY` s before accessing the space with them.

```
public class Form extends OpenSpaces.Entry {
  public org.omg.CORBA.Any asAny() {
    org.omg.CORBA.Any formAny = orb.create_any();
    OpenSpaces.FormHelper.insert(formAny, this);
    return(formAny);
  }
  // ...
}
```

Figure 6.7: Converting to `ANY` in the Java Forms class.

For the reverse transformation of the received ANY s to the concrete forms, the `extract` method of the generated class `FormHelper` is used (see figure 6.10). Before the extraction, however, the agent must test if the operation has been successful. The `SpaceAgent`'s `isNil()` method checks if the result is a remote object reference, which indicates a failure (see section 6.3.3 for explanation). A local object as a result would be the CORBA struct, from which a form can be extracted. Figure 6.8 shows a part of the space agent's Java implementation.

```
public class SpaceAgent {

  org.omg.CORBA.ORB orb;
  OpenSpaces.OpenSpaceInterface hostSpace;

  // check if anAny is a 'pseudo nil'
  protected boolean isNil(org.omg.CORBA.Any anAny) {
    org.omg.CORBA.TypeCode tc =
        orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_objref);
    return( tc.kind() == anAny.type().kind() );
  }

  // Accessing the space
  public org.omg.CORBA.Any read(OpenSpaces.Entry aTemplate) {
    org.omg.CORBA.Any templateAny = aTemplate.asAny();
    org.omg.CORBA.Any backAny = hostSpace.read(templateAny);
    if ( ! isNil(backAny) ) {
      return(backAny);
    }
    return(null);
  }

  // ...
}
```

Figure 6.8: Extract of the `SpaceAgent` class.

Since the generic space agent does not know the exact type of the returned entries, its space accessing operations have to return them as ANY. The extraction takes place at the `MarketAgent`'s trading operations.

6.4.3 Complications with Type Codes

Every type declared in the IDL has a corresponding CORBA `TypeCode`, which provides the type information when extracting a concrete object from a received

ANY. There are constants for all basic types (`tk_xx`, for *type kind*), the constructed types are described with their own code plus the names and codes of all members.

The type code of a generated Java class can be retrieved using its associated helper class. Figure 6.9 shows as an example the struct for the `TailEntry` class from our case studies. On the indicated (*) line a struct type code is created with the `members` array providing the type information of the two fields `index` and `section`.

```
struct TailEntry {                                        // IDL
            long index;
            string section;
};

abstract public class TailEntryHelper {                   // Java

   public static org.omg.CORBA.TypeCode type() {
      if(_type == null) {
         org.omg.CORBA.StructMember[] members =
                     new org.omg.CORBA.StructMember[2];
         members[0] = new org.omg.CORBA.StructMember(
                     "index",
                     _orb().get_primitive_tc(
                                 org.omg.CORBA.TCKind.tk_long),
                     null);
         members[1] = new org.omg.CORBA.StructMember(
                     "section",
                     _orb().get_primitive_tc(
                                 org.omg.CORBA.TCKind.tk_string),
                     null);
         _type = _orb().create_struct_tc(                 // (*)
                     id(),
                     "TailEntry",
                     members );
      }
      return _type;
   }
```

Figure 6.9: The `TailEntry` IDL definition and the construction of its corresponding type code in class `TailEntryHelper`.

With the type code handling of Visibroker we encountered two problems. Both originate in a loss of specific type information.

Use of Wrong Type Code. The IDL mapping of a dictionary is a sequence of (IDL) associations (see figure 6.3 for the declarations). The forms in our examples

use as their sole attribute a dictionary. Thus, the type code of the bindings should be the same as the one of the created `Dictionary` class. To construct a form's type code we use a struct type code – like in the `TailEntry` example above – and add to it as a member a dictionary type code returned by the `DictionaryHelper` method `type()`. Figure 6.10 shows this construction. The indicated line (*) shows the use of the `DictionaryHelper` as described.

```
abstract public class FormHelper {

public static OpenSpaces.Form extract(org.omg.CORBA.Any any) {
  if(!any.type().equal(type())) {
    throw new org.omg.CORBA.BAD_TYPECODE();
  }
  return read(any.create_input_stream());
}

// _type holds the type code of the forms
private static org.omg.CORBA.TypeCode _type;

// lazy initialization with the CORBA TypeCode for forms
public static org.omg.CORBA.TypeCode type() {
  if(_type == null) {
    org.omg.CORBA.StructMember[] members =
                      new org.omg.CORBA.StructMember[1];
    // manually corrected line:
    members[0] = new org.omg.CORBA.StructMember(
                      "bindings",
                      OpenSpaces.DictionaryHelper.type(),   // (*)
                      null);
    _type = _orb().create_struct_tc(
                      id(),
                      "Form",
                      members);
  }
  return _type;
}
```

Figure 6.10: Extract of the generated `FormHelper` class.

However, the originally generated code did not use the dictionary type code, but declared the bindings as a sequence of associations instead. The corresponding line was generated as follows:

```
members[0] = new org.omg.CORBA.StructMember(
                  "bindings",
                  _orb().create_sequence_tc(
                           0,
                           OpenSpaces.AssociationHelper.type()),
                  null);
```

Without the manual correction of the generated code, the bindings of forms sent from the Java client to the space were (correctly) marshalled by the DST ORB as OrderedCollections which led to errors using them.

Loss of Alias Information. A similar problem arose from the use aliased types. An `alias_tc` type code serves to map a new type to an equivalent known type. A typical case where this is used is the Smalltalk class `Symbol` which represents unique strings. Since the IDL does not know any symbol type, the default mapping in DST is: `typedef string Symbol;` With this declaration, the ORB will create an alias type code when marshalling a symbol. Symbols are the natural choice for defining keys in dictionaries. They are unique, i.e. `#xy == #xy` is always true whereas `'xy' == 'xy'` does not hold (`#xy` denotes the symbol „xy"). We use symbols as keys in the bindings dictionary of the forms.

The participation of the Java client in the Market Place led to the problem, that all symbols the space received from the Visibroker ORB were unmarshalled as ordinary strings, i.e. the alias information was lost, the access to the keys led to type errors.

While we could have worked around this problem by adapting the existing code and always use strings as keys for the forms (the uniqueness of the symbols is only really needed with identity dictionaries), we chose to employ the configuration policies for the case. We added to all pre–checking methods a conversion routine, which exchanges all string keys with their equivalent symbol.

6.5 Distributed Event Service

After the discussion of the general CORBA aspects and the experiences with them, we shortly present a further feature of our framework. In addition to the standard tuple space operations, OPENSPACES offers a notification service. A `SpaceAgent` may subscribe to be notified by the space when an entry has been written that matches a given template. This option provides for a semi-synchronous communication style, that does not suspend the agent but immediately informs it of a communication event.

6.5.1 Notification of Space Clients

There are two `OpenSpace` methods to register for notifications. The first, `getNotificationChannelFor: aTemplate`, will cause the space to call the default callback method of the agent, `defaultNotification`, when a corresponding entry arrives. The second uses an additional parameter, `usingSymbol: aSymbol`, which is used to specify the desired callback method. The agent receives as return value an event channel, on which the event will be sent.

The event mechanism is implemented with the CORBA Event Service. The space holds an `EventManager` to care for the event handling. It is called on each writing access to the Space to check if any of the registered subscriptions' templates match with the new entry. If so, it *pushes* an event to the subscriber's CORBA `EventChannel`, which will trigger the subscribing agent's callback method provided on registration. The subscription is terminated when a match is found. For continuous notification it can repeatedly be renewed.

6.5.2 Market Place V.5: Automatic Notification

As example for the convenience provided by the notification service, we implemented an automated variant of the Market Place (see chapter 5). Buyers would like to be informed of newly arriving offers referencing their own requests. Sellers would like to be notified of new requests and also of accepted deals for their own offers.

To avoid repeated manual checking on the client side, we use the notification service. This variant is easily realized by extending the two standard agent classes to `NotifiedBuyer` and `NotifiedSeller`. Upon every request the buyer makes, he subscribes to the notification service of the space with a template of an offer containing the corresponding request number. When notified, he automatically updates his offers collection with the newly arrived offer. The seller takes similar action for new deals, for requests she subscribes at initialization time.

Chapter 7

Conclusions

For this thesis we investigated how the tuple space model can be extended to be *configurable* in its behavior, allowing it to be adapted to the needs of an application. It should provide the standard features for tuple style communication and be easily *extensible* to provide means for integrating new functionality. To be usable in *heterogeneous* distributed systems the medium must be accessible for any external client.

We introduced OPENSPACES, an object-oriented framework for coordination, which aims to fulfill the above requirements.

7.1 Need for Configurability and Heterogeneity

The construction of complex modern applications, built from several autonomous components, that may be distributed over the network, requires specialized models for the *coordination* of the collaborating processes. The tuple space model offers many advantages for this task. Since the proposal of the original LINDA system many variants have been introduced, each extending the base model with new features.

Despite - or because of - its elegance and simplicity, the tuple space models often lack of flexible *abstractions*. It would be highly desirable if a tuple space model would allow a programmer to establish protocols at more abstract levels, to combine typical series of access operations, or to delegate low level „housekeeping" tasks to the medium.

A few models have introduced means to modify a tuple space's behavior. They feature *reactivity* of the space, allowing several actions to be triggered on access events. These models, however, are bound to certain limitations. They are only

usable with their provided internal agents. The implementation language of both the spaces and the clients is Prolog, which complicates their integration into existing applications and environments. Also, they do not provide the benefits of the object-oriented programming paradigm.

To be used in open distributed systems, a coordination model must fulfill the following *heterogeneity* requirements: The model must provide access for *external agents*. The agents may be written in *any programming language* that suits their needs best. The medium must run under *multiple platforms*. Clients may run each on its (potentially different) platform.

7.2 OPENSPACES's Solutions

The OPENSPACES framework provides three main axes of *configurability*. First, simple subclassing of the core classes provides for *white-box extension* and reuse. Second, the specialization of *configuration policy* objects offers means to specify the behavior of the coordination medium in a wide range of possibilities. Third, besides compile-time extensions, the configuration policies provide for run-time *reconfiguration* of a given setup.

The following sections present these three aspects of the framework's variability and refer to their realization in our case studies.

7.2.1 White-Box Extensions

Extending a framework in a white-box style requires knowledge of its internals [JF88]. The object-oriented paradigm permits programming-by-difference, the use of inheritance lets subclasses define but the new functionality. Three classes of OPENSPACES are candidates for extensions by subclassing:

`SpaceAgent`. Subclassing `SpaceAgent` helps to specialize the client for handling application specific tasks and abstract away the underlying communication structures. For our case study we subclassed the standard space agent to `MarketAgent`, `Buyer`, and `Seller`, plus the variants `AssertedBuyer`, `LightBuyer`, `NotifiedBuyer`, and their seller counterparts.

`OpenSpace`. If an application needs new or specialized operations, they can be added by subclassing the space abstraction `OpenSpace`. An example is the mentioned `rhonda` operator as introduced in TSPACES (see 3.3.6). Also bulk-retrieving operations could be added, which define a range of how many entries should be returned, as realized in OBJECTIVE LINDA (see 3.3.2). Another operation which has been discussed is an update primitive, allowing an agent to e.g. increment a counter directly at the space instead of the series of `take` - `incre-`

`ment - write`. Many extensions however can be realized with suitably chosen configuration policies, as shown with the index handling example in 5.4.

`Entry`. Subclassing the `Entry` class serves two main purposes: (1) to define application specific data fields like the `Form` in our case study (see 5.2), and (2) to map the entry type to a specific configuration policy as shown in the consistency assertions example in 5.3.

7.2.2 Defining Configuration Policies

The `ConfigurationPolicy` subclasses define the semantics of the coordination medium. Each entry type must be registered at a space before it can be used. This registration maps the entry's class to a policy object. This policy defines the matching algorithm and offers hook methods that can be used to fully control each space access.

Matching Algorithm. The strategy for the associative lookup of entries is simple: the space scans its entries collection for an entry that matches the given template. The boolean function to test for a match is provided by the policy object. An example is shown in figure 5.4 with the FormMatching. The function may test any condition, special values may be required, a keyed matching may be performed, like in our locking example 5.7.

Pre- and Post-Access Hooks. For each access operation, the `ConfigurationPolicy` class defines two *hook* methods that are invoked before and after the actual space operations, respectively. The hooks that are called before may validate the used entry or template. They also may modify it or reject it at all. Those hooks called after the space operations have the occasion to validate the found entry before they are delivered to the calling agent.

Besides validating the used entries, the hook methods may perform any additional actions supporting the employed protocol. They may access the space on their own, to check or re-establish its consistency. They may also perform external activities like accessing files. See table 5.1 for an overview of the tasks performed by the configuration policies in our case studies.

7.2.3 Run-Time Configuration

In addition to the static extension of the framework, OPENSPACES provides for dynamic configuration of the spaces. Indeed, changing requirements of any kind can necessitate a reconfiguration of the policies. With OPENSPACES, an application can change the policies dynamically on a running system.

Dynamic Configuration. Once defined, the configuration policy objects may be used in a black-box style by registering them at a space. Since the concerned configuration policy is looked up for every space access, a modification of the mapping automatically becomes effective. Therefore, it is easy to apply a new configuration on the fly. It is actually the standard procedure to register all needed entries after the creation of the space. The registration methods use the same mechanism for mutual exclusion as the basic operations. This guarantees, that no running execution is interrupted.

Reconfigurations. When reconfiguring a running space on the fly, it will often be the case that some instances of the affected entry classes are already in the space. Depending on the new policy it may be necessary to adapt those to the new protocol. The class `ConfigurationPolicy` offers for this end a special hook method, which is called right after each registration of an instance. This method gives a programmer the possibility to specify all necessary actions for a clean transition. Figure 5.12 shows an example of its use.

After this overview of the configurability options of OPENSPACES, the next section summarizes its heterogeneity features.

7.2.4 Heterogeneity

OPENSPACES aims to be usable for coordination in heterogeneous environments, which is an important requirement for open distributed applications. It fulfills this claim in several concerns.

Server Platforms. The framework is written in Visual Works Smalltalk, which runs under four major platforms: Solaris, Windows, Linux, Macintosh. The code is fully portable, like this an `OpenSpace` may work on each of them.

Client Platforms. OPENSPACES features distributed access of the coordination medium. Clients may reside anywhere on the Internet, communication between the spaces and their clients is implemented with CORBA on top of TCP/IP. This means that the client platforms are not subject to restrictions. For the provided `SpaceAgent` the same holds as for the server platforms.

We tested several combinations of platforms for the spaces and the client agents. The only restriction we encountered, was the impossibility to run both a space and a client on the same Windows 9x or Macintosh 9.0 computer. This seems to be a problem with the local TCP/IP loop.

External Clients. By using CORBA as its communication layer OPENSPACES opens its services to any client implementing the interface defined in its IDL module. There are no other presumptions about the nature of the agents.

Client Languages. The programming language an agent may be written in is restricted only by the requirement to dispose of an ORB. Since the Since CORBA is a standard ported by a large number of vendors, there are implementations on virtually every operating system/programming language combination.

Our Java client presented in 6.4 is a successful test of the latter two heterogeneity claims. Together with the two platform related claims, these properties give a good basis for a heterogeneous coordination platform.

7.3 Perspectives

We see mainly two lines of possible future work on OPENSPACES: one concerning extension towards a general service structure, the other concerning practical improvements.

General Service Structure. To establish a general OPENSPACES-Service, it would be interesting to add possibilities to not only dynamically register policies but also new entry types. This would allow clients to fully set up an application's protocol without restarting the space to add the new entry classes.

We made experiments with dynamically adding new IDL to the repository. This is possible, the definitions can become active as soon as registered. We then need the corresponding classes in the Smalltalk image. With Smalltalk it is easily possible, to dynamically add a class and add instance variables to it. On the registration of the IDL definition of the new entry type, we need to parse the definition and add a corresponding class to the image. It is a bit more complex with the configuration policies.

While the entries can be defined in terms of their IDL structs, the configuration policies have to be defined in Smalltalk, which restricts the universality of the service. One – non-trivial – solution would be to define a special language for the policy definitions. Another solution is to define a collection of generic configuration policies, that can serve for several entry types. However, the benefits of mass-tailored policies cannot be fully exploited with generic policies.

Practical Improvements OPENSPACES is a prototypical framework, to be employed in a practical context, some improvements would have to be scheduled. The code is not optimized for performance whatsoever. It would be desirable to add a transaction handling mechanism. We experimented with the CORBA Transaction Service, which could be a basis for this. Also a general timeout mechanism should be considered.

Bibliography

[COORD96] P. Ciancarini and C. Hankin, editors. *Proceedings of COORDINATION'96: Coordination Languages and Models*, number 1061 in LNCS. Springer-Verlag, 1996.

[COORD97] D. Garlan and D. Le Métayer, editors. *Proceedings of COORDINATION'97: Coordination Languages and Models*, number 1282 in LNCS. Springer-Verlag, 1997.

[COORD99] P. Ciancarini and A. L. Wolf, editors. *Proceedings of COORDINATION'99: Coordination Languages and Models*, number 1594 in LNCS. Springer-Verlag, 1999.

[COORD00] A. Porto and G.-C. Roman, editors. *Proceedings of COORDINATION'00: Coordination Languages and Models*, number 1906 in LNCS. Springer-Verlag, 2000.

[Arb96] F. Arbab. The IWIM model for coordination of concurrent activites. In Ciancarini and Hankin [COORD96], pages 34–55.

[BK96] J.A. Bergstra and P. Klint. The toolbus coordination architecture. In Ciancarini and Hankin [COORD96], pages 75–88.

[BOV99] C. Bryce, M. Oriol, and J. Vitek. A coordination model for agents based on secure spaces. In Ciancarini and Wolf [COORD99], pages 4–20.

[CDK94] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems Concepts and Design*. Addison-Wesley, 1994.

[CG90] N. Carriero and D. Gelernter. *How to Write Parallel Programs: a First Course*. MIT Press, Cambridge, 1990.

[Cia99] P. Ciancarini. Coordination models ans languages for agents. University of Bologna, 1999.

[COZ00] M. Cremonini, A. Omicini, and F. Zambonelli. Coordination and access control in open distributed agent systems: The TuCSoN approach. In Porto and Roman [COORD00], pages 99–114.

[CR96] P. Cinacarini and D. Rossi. Jada: Coordination and communication for java agents. In *MOS'96: Towards the Programmable Internet*, LNCS 1222, pages 213–228, Linz, Austria, 1996. Springer-Verlag.

[DNO97] E. Denti, A. Natali, and A. Omicini. Programmable coordination media. In Garlan and Métayer [COORD97], pages 274–288.

[DO99] E. Denti and A. Omicini. Engineering multi-agent systems in LuCe. In S. Rochefort, F. Sadri, and F.Toni, editors, *ICLP'99 International Workshop on Multi-Agent Systems in Logic Programming*, Las Cruces (NM), 1999.

[FHA99] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces: Principles, Patterns and Practice*. Addison-Wesley, 1999. ISBN: 0201309556.

[GC85] D. Gelernter and N. Carriero. Generative communication in linda. *ACM TOPLAS*, 7(1), 1985.

[GC92] D. Gelernter and N. Carriero. Coordination languages and their significance. *Comm. ACM*, 35(2), 1992.

[Gel89] D. Gelernter. Multiple tuple spaces in linda. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89: Parallel Architectures and Languages Europe. Volume II: Parallel Languages*, number 366 in LNCS, pages 20–27. Springer-Verlag, 1989.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[Hup90] S. Hupfer. Melinda: Linda with multiple tuple spaces. YALEU/DCS/RR-766, Yale University, 1990.

[JF88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[Kie97] T. Kielmann. *Objective Linda: A Coordination Model for Object-Oriented Parallel Programming*. Ph.D. thesis, University Siegen, 1997.

[Lea96] D. Lea. *Concurrent Programming in Java, Design Principles and Patterns*. Addison Wesley, Reading, MA, 1996.

[Lum99] M. Lumpe. *A Pi-Calculus Based Approach to Software Composition*. Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, 1999.

[MC94] T. W. Malone and K. Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1), 1994.

[MK88] S. Matsuoka and S. Kawai. Using tuple space communication in distributed object-oriented languages. In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, pages 276–284, 1988. Published as Proceedings OOPSLA '88, ACM SIGPLAN Notices, volume 23, number 11.

[ML95] N. Minsky and J. Leichter. Law-governed linda as a coordination model. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 125–146. Springer-Verlag, 1995.

[Mon99] T. Montlick. *The Distributed Smalltalk Survival Guide*. SIGS books, Cambridge, 1999.

[Nie93] O. Nierstrasz. Composing active objects. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 151–171. MIT Press, 1993.

[ODN95] A. Omicini, E. Denti, and A. Natali. Agent coordination and control through logic theories. In M. Gori and G. Soda, editors, *Topics in Artificial Intelligence – Proceedings of the 4th AI*IA Congress of the Italian Association for Artificial Intelligence*, volume 992 of *LNAI*, pages 439–450, Firenze, 1995. Springer-Verlag.

[OMG98] OMG, Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.2*, 1998.

[OTZK01] A. Omicini, R. Tolksdorf, F. Zambonelli, and M. Klusch, editors. *Coordination of Internet Agents: Models, Technologies and Applications*. Springer-Verlag, 2001.

[PA98] G. Papadopoulos and F. Arbab. Coordination Models and Languages. In *The Engineering of Large Systems*, volume 46 of *Advances in Computers*. Academic Press, 1998.

[PWGB99] D. Pedrik, J. Weedon, J. Goldberg, and E. Bleifield. *Programming with Visibroker, a Developer's Guide*. Wiley Computer Publishing, New York, 1999.

[RW96] A. Rowstron and A. Wood. Solving the linda multiple rd problem. In Ciancarini and Hankin [COORD96], pages 357–367.

[RW97] A. Rowstron and A. Wood. Bonita: A set of tuple space primitives for distributed coordination. In *Proceedings of HICSS-30*, volume 1, pages 379–388. IEEE Computer Society Press, 1997.

[Sci95] Scientific Computer Associates, New Haven. *Paradise, User's Guide and Reference Manual*, 1995.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[Tol97] R Tolksdorf. Berlinda: An object-oriented platform for implementing coordination languages in java. In Garlan and Métayer [COORD97], pages 430–433.

[WMLF99] P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford. T spaces. *IBM Systems Journal*, 37(3), 1999.