Coordination in Open Distributed Systems

Robert Tolksdorf

Reproduction of the printed version of 1995

Bismarckstr. 18
14109 Berlin

Robert Tolksdorf

Digital unterschrieben von Robert Tolksdorf
DN: c=DE, cn=Robert Tolksdorf, givenName=Robert Vinzenz, sn=Tolksdorf, serialNumber=DTRWM685137337394922, title=Dr.
Datum: 2018.03.30 18:54:16 +02'00'

Coordination in Open Distributed Systems

Robert Tolksdorf

http://www.robert-tolksdorf.de

Contents

1 Introduction: A view on coordination in computing systems 4

2 Coordination focussing on data-exchange: LINDA 10

2.1 Linda's tuple-space and its operations 11
2.2 Two examples of Linda-programs 13
2.3 Tuple-space predicates and multiple tuple-spaces 14
2.4 Discussion 15
2.5 Bibliographic remarks 16

3 Coordination focussing on process synchronization: ALICE 21

3.1 Tuples and fields in Alice 22
3.2 Agents in Alice 24
3.2.1 Local agent-spaces 27
3.3 Process synchronization with Alice 29
3.3.1 Coordinating conditional execution 29
3.3.2 Coordinating loops 30
3.3.3 Synchronization of groups of agents 31
3.4 Turing machines with Alice 32
3.5 Historical and bibliographic remarks 34

4 Coordination of services in open distributed systems: LAURA 38

4.1 Design motivation 38
4.2 Identification of services 43
4.3 On naming in open distributed systems 46
4.4 Laura's operations 48
4.5 Bibliographic remarks 53

5 Prescribing LAURA formally 55

5.1 A type system with subtyping 56
5.1.1 Rules for type-equivalence 56
5.1.2 Rules for subtyping 57
5.1.3 The semantics of Laura's service-type definitions 59

5.2 A formal model of coordination: The Bag-Machine 61
5.3 Technical preliminaries for the Bag-Machine 64
5.3.1 Labeled event structures 64
5.3.2 Open labeled event structures 66
5.4 A labeled event structure for the Bag-Machine 68
5.5 The behavior of agents using the Bag-Machine 69
5.6 Embedding coordination and computation languages 74
5.7 Example: Specifying Linda with the Bag-Machine 76
5.8 The semantics of Laura's operations 79
5.9 Bibliographic remarks 84

6 An experimental implementation of LAURA 86

6.1 A C-Embedding: C-Laura 87
6.2 A csh-embedding: csh-Laura 88
6.3 The STL-precompiler 89
6.4 The Laura-library 91
6.5 The Bag-Machine instance 91
6.6 A distributed Bag-Machine 94
6.6.1 Protocols used for the distributed Bag-Machine 98
6.6.2 Protocols for joining and leaving nodes 100
6.6.3 Extended Bag-Machine-organizations 104
6.7 Experience with the prototype 106
6.8 Bibliographic remarks 106

7 Outlook and perspectives 109

8 Acknowledgments 112

Appendix

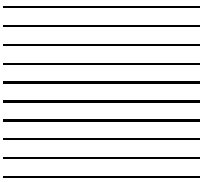
A LAURA's subtyping tested by ALICE-agents 114

A.1 Testing records 115
A.2 Testing operation signatures and interfaces 117
A.3 Example executions 118

B Bibliography 120

Chapter 1

Introduction: A view on coordination in computing systems



Computer science and information technology undergo a significant change in what paradigm guides the organization of computing systems: Isolated or centralized machines are rapidly replaced and integrated by networked, distributed computing systems. The scale of distribution is immense – it crosses organizational structures and continents. Network and telecommunication technology is being installed that transforms computing systems from locally available equipment dedicated to specific purposes into universal access points to information from all over the world allowing users to perform tasks that are beyond the computation potential of the machine used. "Information-superhighways" and their utilization are often expected to have a social impact that is sometimes compared to the invention of print or even fire.

The technological development directs a spotlight on the questions concerning the coordination of activities in such systems. How should they be structured; what paradigms should guide communication and synchronization in these systems; how should services be provided and used?

The systems can be discussed using the following model of cooperating agents[1], the *agent-world* as a conceptual basis. In the agent-world agents act in the context of some overall goal, reflected by some notion of benefit for agents. The concepts of the agent-world are as follows:

- **Agent** An active entity in the agent-world.

[1] The term "agent" here should not be confused with its understanding in the AI-community as in blackboard systems.

- **Action** Something that happens.
- **Process** A sequence of actions executed by an agent.
- **Synchronization** Actions concerning the start, blocking or unblocking and the termination of processes.
- **Communication** Actions concerning the exchange of data amongst agents.
- **Service** Results from an interaction of agents distinguished as service-user and service-provider. A service is of benefit to the service-user by utilizing the results of activities of the service-provider. User and provider share an agreement on communication of data and synchronization of processes.

Instances of the agent-world can be found in many fields of computer science. An imperative program, for example, is an instance of our model in that procedures are agents as defined by the program-text of the procedure-body. A procedure-call involves the communication of arguments and results and starts a process by the execution of the procedure. The calling procedure uses the service provided by the called procedure to achieve some goal. Both share an agreement that includes rules on parameter-passing mechanisms, such as stack-usage conventions, and data-representation, for example.

However, the agent-world model is more suited to describe systems in which agents are spatially dispersed and act autonomously. An example for such a system could be that of a distributed object system. Here, objects can be considered agents whose actions are defined by the implementation of methods. Processes then are executions of methods and communication means sending a message to an object. Synchronization actions cover the remaining mechanisms in object invocation. A service occurs when an object invokes methods of another object, where the invoker is the service-user and the invoked object the provider. The agreement includes, as an example, that only methods offered at some object interface are invoked or that invocation at all is admitted.

Our model focusses on the concerns involved in process synchronization, in communication and in how services are coordinated. It abstracts from the outform of activities in the execution of processes. We use the term *coordination* to refer to actions for synchronization, communication and service-usage and -provision and the term *computation* for actions of processes. In this thesis, we focus on coordination solely and deal with it separated from computation.

The separation does not mean that computation and coordination are independent or dual concepts. In contrast, they are orthogonal in the sense that they are two dimensions under which an agent-world can be discussed separatly but which do not provide a complete view on a system. The imcompleteness becomes obvious in that it makes no sense to provide a service without communicating the results to the service-user. On the other hand, coordination make no sense if there are no activities to be coordinated.

Coordination has several aspects such as efficiency in terms of speed or suitablitiy for specific purposes such as multi-media requirements. We focus on the aspect of linguistic means to express the concerns of coordination as well as on their implementation. Given the conceptual separation of coordination and computation, we deal with *coordination languages* as opposed to *computation languages*. We do not attempt to design parallel

or distributed programming languages, but take the view that the separation of concerns leads to two families of languages.

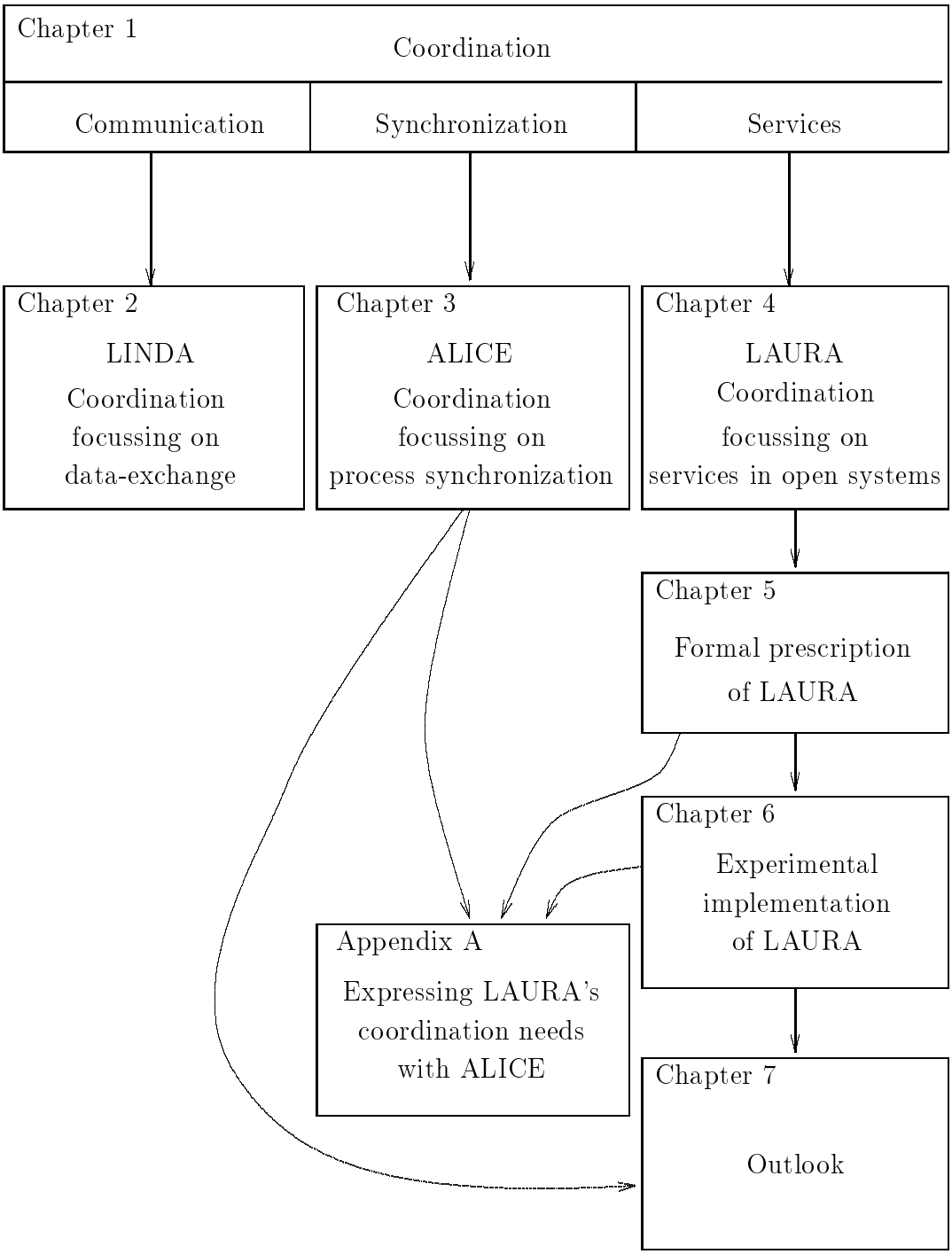


Figure 1.1: The structure of this thesis

The intention with this thesis is to verify this conception against the problem of coordinating services in open distributed systems. We do so by reviewing and designing coordination languages that provide linguistic means to express coordination actions.

The concerns we identified above with the term coordination will be represented in this thesis by three coordination languages. First, we discuss the language LINDA – developed at Yale University starting in the early 1980s – and then present ALICE, a design of our own. LINDA can be taken as a coordination language that focusses on the communication aspect of process coordination while ALICE puts more emphasis on the synchronization aspect. As the main contribution of this thesis, we then lay out the design and implementation of a coordination language for services in open distributed systems, called LAURA.

It shows that the separate consideration of coordination is an enabling conceptual basis to meet requirements of open distributed systems, such as heterogeneous architectures and the support of interoperability amongst programming languages.

The choice of an uncoupled communication style proves to be adequate to cope with the dynamics of open systems. By an experimental implementation of the language LAURA we show that its design is implementable and has several advantages.
At least two lines of research provide a context to our work:

- First, the intention of the LINDA-group as outlined in [Gelernter, 91] meets our interest in coordination as put in the agent-world and has put the first conceptual basis of separating coordination and computation and of focussing on the design of coordination languages. However, the achievement of LINDA is to provide a coordination language for parallel systems. Our approach is novel in that we focus on open systems which impose quite different requirements.

- Open distributed systems have been the focus of the standardization efforts undertaken in the ISO which have resulted in the standard on open distributed processing, ODP [ISO/IEC JTC1/SC21/WG7, 93a]. ODP provides a model which is to be instantiated by a number of major computer industry vendors being members of the object management group consortium OMG ([OMG91], [OMG92]). An example for such a concrete industrial product is IBM's DSOM ([IBM93]). Our approach contrasts these developments as they neither employ a conceptual basis of separated focus on coordination nor provide a formal basis for their solutions.

Figure 1.1 shows the structure of this thesis and the connections amongst the chapters. The model of coordination given in this chapter serves as a structure for this thesis. Our focus is on the coordination of services in open distributed systems, which is reflected by the detail in which we investigate in this aspect.

- In chapter 2 we focus on the communication aspect of coordination. We do so by reviewing the coordination language LINDA which introduces a tuple-space and operations that manipulate it. We describe them and give examples for LINDA-programs. We review features such as tuple-space predicates and multiple tuple-spaces as found in the different versions of LINDA. Finally, we discuss why LINDA's concepts are of major importance when investigating in the communication aspect of coordination.

- LINDA does not contain a model of process-synchronization but inherits it from some host-language. In chapter 3 the synchronization aspect of coordination is our focus. We introduce our coordination language ALICE which includes a process model and detail out its design. We then demonstrate how synchronization problems from imperative programming can be dealt with in a concurrent coordination language such as ALICE on a very fine grain. We show that the separated concept of coordination is orthogonal to computation by defining agents in ALICE that are capable of interpreting Turing-machines.

- The coordination of services is the topic of the following chapters. In chapter 4 we introduce our coordination language LAURA. First, we lay out its design rationale and discuss the issues of identification of services and of naming in open distributed systems. An informal description of LAURA's operations and how they work concludes this chapter.

- An informal description requires a formal prescription that identifies correct implementations. For LAURA it is presented in chapter 5. We use typed interfaces for the identification of services in LAURA. We formalize the notations by defining a type system which takes into account the considerations on naming we presented before. The type system includes a subtyping relation and gives semantics to interface definitions of services.

- The coordination languages in this thesis all make use of the manipulation of multisets of some elements. We formalize the mechanism by introducing an understanding of multisets which uses ϵ-structures to talk in a uniform framework on elements of a multisets and their multiple instances. We then give a true concurrent definition of a process called the Bag-Machine that implements the manipulation of multisets by labeled event structures. We extend these structures to allow us to define the behavior of agents that make use of the Bag-Machine.

- We demonstrate this mechanism with a formal definition of LINDA. Then, we define the semantics of LAURA by expressing its operations in terms of the Bag-Machine.

- After this formal prescription, we demonstrate that LAURA is implementable by an experimental prototype. It includes embeddings in two programming languages, a pre-compiler and a library. We lay out an architecture for a distributed implementation and describe its protocols. We discuss how our architecture can be extended and what experiences demonstrated the well-suitability of our approach for a solution to the coordination problem in open distributed systems.

- To conclude, we give an outlook on the use of our coordination model and the languages ALICE and LAURA. We identify further issues such as directions towards a more general model of the design, application and exploitation of open distributed systems in chapter 7.

- To round off, we show in appendix A how the implementation of LAURA has coordination needs that can be satisfied by ALICE. In particular, we define ALICE-agents that perform tests on the subtyping mechanism required for LAURA.

Berlin, October 1994

Chapter 2

Coordination focussing on data-exchange: Linda

Having presented our model of coordination in the introduction, we review in this chapter the coordination language Linda which puts special emphasis on the communication aspect of coordination in parallel systems.

Linda is a language for coordination in parallel systems that has been studied from about the midst-eighties. The underlying view of a parallel system is that of an *asynchronous ensemble*, in which all work in the system is performed by *agents*. A set of agents forms an *ensemble* by coordinating their activity asynchronously via some media. The actual work they perform is carried out independently, asynchronously and autonomously.

Linda introduces an *uncoupled* communication paradigm which is based on the abstraction of a *tuple-space*. It acts as a shared store of data which is kept as *tuples* that are addressed associatively by a pattern used in a matching-mechanism to select a tuple. It is unknown, which agent put the tuple into the tuple-space, thus communication partners remain anonymous to each other.

We chose Linda as the language reviewed for the communication aspect of coordination for the following reasons. First, Linda puts emphasis on coordination only. This separation of concerns leads to simplicity and does not introduce concepts that are beyond the scope of this thesis.

Linda takes an abstract view on coordination on a high level. It does not imply restrictions on concrete implementations and thus allows it to discuss the communication aspect on a conceptual level.

The coordination mechanisms has been proven to be fundamental and powerful compared to other mechanisms ([Polze and Löhr, 92]). It has been shown that well known communication paradigms can be emulated by LINDA ([Carriero and Gelernter, 89a]) thus making LINDA a worthful candidate for a discussion of the communication aspect of coordination.

This chapter is organized as follows: First, we give an overview on the LINDA tuple-space and its operations in section 2.1 and illustrate it with two examples in section 2.2. In section 2.3 we describe additional operations of an earlier version of LINDA and some extensions of the latest version. We discuss LINDA in section 2.4 in the context of our agent-world and give pointers to LINDA-related work in section 2.5.

2.1 LINDA's tuple-space and its operations

LINDA is a coordination language for parallel programming which was first introduced by David Gelernter in his doctoral thesis ([Gelernter, 82]) and presented to a wider audience in [Gelernter, 85]; one of the first implementations is described in [Carriero and Gelernter, 86]. In the following, we give an informal outline of this coordination language; the bibliographic remarks give pointers to further details and to the family of LINDA-like languages that has been stimulated by the work at Yale University.

In LINDA, coordination is performed by operations on an intermediate medium, called the *tuple-space*. Agents perform either computation expressed in some programming language or coordinate with other agents using LINDA's tuple-space operations. Thus, a LINDA-system is a combination of a conventional programming language with the LINDA operations, called a LINDA *embedding*.

The tuple-space is an unordered collection of *tuples* that are generated and consumed by agents. A tuple is an ordered set of *tuple-fields* notated in angle brackets. What can be used as a tuple-field depends on the embedding. In the following, we assume a C-LINDA, meaning the programming language C with the LINDA-operations embedded. We use a syntax which neglectable differs from that of the Yale-implementation[1].

Examples of tuples in such an embedding are `<"Robert",10.4>` or `<'c',TRUE,20>`. The first is a two fielded tuple, consisting of a string-field with the value `Robert` and a floating-point field valued 10.4. The second example consists of a character-field with value `c`, a boolean-field of value true and a 20 in an integer field. Fields that have a value as their contents are called *actuals*.

Tuples can be emitted to the tuple-space using LINDA's `out`-operator. A statement that puts the first example-tuple into the tuple-space is `out(<"robert",10.4>);`. When the tuple `out`-ed is constructed and passed to the tuple-space, the agent executing it continues with its operations without blocking.

In a LINDA-embedding the values can also be determined by resolving a binding of a program-variable. Thus, if `i` is declared as an `int` and has currently the value 20, then `out(<'c',TRUE,i>)` puts the tuple `<'c',TRUE,20>` into the tuple-space.

[1] We use an explicit notation for tuples (`out(<"abc",10>)`) instead of adopting the C-syntax for argument-lists (`out("abc",10)`) and refer to types (e.g. `bool`) that are not implemented in C-LINDA.

References to variables are always resolved and never exist in the tuple-space, only within the agent.

Tuples can be retrieved from the tuple-space by an agent performing an `in`-operation. It has to provide a *template* as an argument which is a tuple-pattern used for a *matching*-mechanism. This mechanism identifies tuples that match the template and selects one for withdrawal from the tuple-space.

A template is an ordered collection of tuple-fields just like a tuple but with the exception that some fields can be filled with *formals* that act as place-holders for any value of a specific type. An example of a template is `<'c',?bool,?int>`. Here, the first template-field is an actual character with the value `c` and the following two fields are formals, marked with the tag `?`. The first is a placeholder for any value of the type `bool`, the second one for any value of type `int`.

The matching of a template and a tuple is guided by the *matching-relation*. A tuple t and a template t' are matching, if the following conditions hold:

1. t and t' have the same number of fields.

2. Each field from t matches the corresponding field from t'.

3. Any actual matches an actual with equal value.

4. Any formal matches an actual of the same type.

So the operation `in(<'c',?bool,?int>)` can retrieve the second example tuple above, but not the first. An agent that performs an `in` is blocked until a matching tuple is entered to the tuple-space by another agent performing an `out`. If multiple matching tuples are found, one is chosen non-deterministically.

In a LINDA-embedding, the values of fields of the retrieved tuple can be bound to program variables. If the variables `b` and `i` are declared to be variables of type `bool` and `int`, resp., then after retrieving the above tuple with `in(<'c',?b,?i>)`, `b` would have the value `TRUE` and `i` the value 20.

The third LINDA-operation `rd` is identical to `in` with the exception that the tuple matched is not removed from the tuple-space. All synchronization and communication amongst agents is performed by depositing and retrieving tuples to and from the tuple-space using `out`, `in` and `rd`.

Given that multiple agents perform these operations concurrently, choices have to be made about which specific tuple is selected in the matching process and which agent is unblocked by returning it. LINDA does not make any guarantees on fairness of these choices. It leaves the possibility open that a tuple is held forever in the tuple-space and never selected and that an agent is treated unfair by preferring other agents in the unblocking.

The fourth LINDA-operation `eval` is used for the creation of parallel activity. It takes an *active tuple* as an argument. Here, the fields can be actuals or references to functions. The non-blocking `eval`-operation starts the evaluation of these functions in parallel. Their result values then replace the function-references as actuals. When all fields have been evaluated, the resulting tuple is put into the tuple-space.

Let as an example, the function `square` be defined as taking an integer as parameter and resulting in an integer which is the parameter multiplied with itself. Then the operation `eval(<a,square(a),b,square(b)>)` starts the evaluation of `square(a)` and `square(b)` in parallel to the continuing activity of the agent performing the `eval`. Let `a` have the value 2 and `b` the value 3, then the tuple `<2,4,3,9>` is finally put into the tuple-space.

2.2 Two examples of LINDA-programs

To give an impression of how LINDA-programs look like, Figure 2.1(a) shows a C-LINDA program for the well-known dining philosopher's problems, taken from [Carriero and Gelernter, 89b]. It is demonstrating the synchronization aspect of LINDA's constructs.

```
phil(i)
  int i;
{
  while(1) {
    think();
    in("room ticket");
    in("chopstick", i);
    in("chopstick", (i+1)%Num);
    eat();
    out("chopstick", i);
    out("chopstick", (i+1)%Num);
    out("room ticket");
  }
}

initialize()
{
  int i;
  for (i=0; i < Num; i++) {
    out("chopstick", i);
    eval(phil(i));
    if (i < (Num-i))
      out("room ticket");
  }
}
```

(a) Dining philosophers

```
master()
{
while (get(new_sequence)) {
  out("task", new_sequence);
  if (++tasks> HIGH_WATER_MARK)
    do {
      in("result",?result);
      update_best_results_list(result);
    } while (--tasks >LOW_WATER_MARK)
  }
while (tasks--) {
  in("result",?result);
  update_best_results_list(result);
  }
  report_results();
}

searcher()
{ {
  do {
    in("task",?seq);
    value=compare(seq,target);
    out("result", value);
  } while (search_is_not_complete())
}
```

(b) DNA-sequence matching

Figure 2.1: Examples in C-LINDA

`initialize` creates `Num` tuples to model the chopsticks and `Num` philosopher agents with `out` and `eval`, resp. Also, it generates `Num-1` room-tickets as tuples with `out` which equals the number of philosophers that can try to eat without deadlocking. A philosopher agents tries to get a room ticket with `in` which allows him to get the chopsticks with `in`. The chopstick tuples are numbered, thus the notion of the "left" and "right" chopstick is encoded in that index. After eating, the agent releases the chopsticks and its ticket with `out`. The example shows, how a semaphore-oriented synchronization is coordinated with LINDA[2].

Another example that combines synchronization and communication is shown in figure 2.1(b), which is a parallel DNA sequence search. There is one agent `master` that has to find a list of best matching DNA sequences stored in a database with respect to a new sequence. It uses a couple of `searcher` agents that are initialized with the new sequence in `target`.

Each searcher waits for a request to perform a comparison of the initialized sequence `target` with another by doing an `in`. The sequence to be compared is read into a formal field. The result of the comparison is emitted as a tuple with an `out`.

The master scans linearly through the database and `out`'s for each entry a tuple with the database sequence. The number of outstanding tuples is limited by `HIGH_WATER_MARK` and `LOW_WATER_MARK`. As the searchers are activated by the presence of a matching tuple, the comparisons are performed. The master collects the results with `in` and produces a list containing the best comparison values of the new sequence with those stored in the database. Note that only the quantity of the similarity of the new sequence to those in the database is collected and the best matching sequence cannot be told from the result.

2.3 Tuple-space predicates and multiple tuple-spaces

The initial LINDA-design had two additional predicates, `inp` and `rdp` ([Carriero and Gelernter, 89a]). They were defined as true, if a tuple matching a given pattern was available in the tuple-space. In addition, in this case the binding of values to formals was performed so that the operations are rather non-blocking versions of `in` and `rd` than predicates. Both operations were later dropped because their semantics was unclear and their practical use was considered doubtable.

[Gelernter, 89] describes the final component of LINDA in its current definition, *multiple tuple-spaces*. It introduces a new type, a fifth operation and a naming scheme to the language. The idea is to make tuple-spaces first class objects in the tuple-space. Thereby the flat structure of a single global tuple-space is replaced by a hierarchy of tuple-spaces.

The new type is called `ts` and identifies a tuple-space. The embedding has to take care of the binding of a value of type `ts` to program variables. The only operation that can generate a tuple-space is `tsc` for tuple-space-create. It is evaluated with an `eval` and results in a new tuple-space. It is located in the "current tuple-space",

[2] [Carriero and Gelernter, 89b] point out that this example can run into livelocks because of different speeds of agents and because of the lack of a notion of fairness in LINDA.

meaning the tuple-space in which the agent executing the `tsc` operates. By executing `eval(tsc(Q))`, a new and empty tuple-space is created and can be referenced by the name `Q`.

LINDA-operations can be prefixed by a name that references a tuple-space. Thus, `Q.out(<"Robert",10.4>)` inserts the tuple into the newly created tuple-space `Q` instead of the global tuple-space. The hierarchy of tuple-spaces can be accessed by a hierarchical naming scheme, where tuple-space names are combined with `/`, and names can be relative to the current space or absolute to some root. After performing `Q.eval(tsc(P))`, the newly created tuple-space is accessed by `Q/P.in(<"Robert",?float>)`. Given that `Q` was created in the tuple-space `R` located at the top of the hierarchy, it can be accessed by `/R/Q/P.in(<"Robert",?float>)`. Figure 2.2 illustrates this situation.

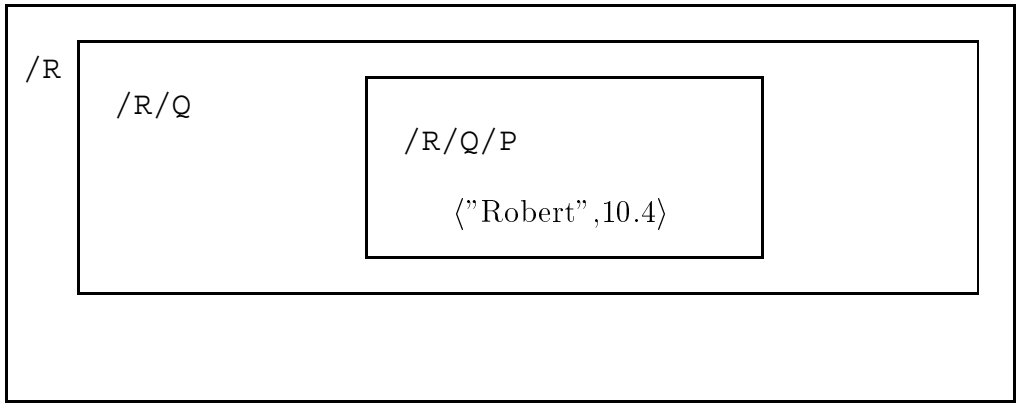


Figure 2.2: Multiple tuple spaces `Q`, `R` and `P`

Tuple-spaces themselves are represented as tuples. Thus the LINDA operations can be applied to them. If `q` is a program variable of type `ts`, then the operation `in(<?q>)` will search for a tuple representing a tuple-space. It then removes it completely and binds an image of the tuple-space to `q`. If there are active tuples under evaluation at the time such an `in` occurs, these are freezed and restarted when an `out(q)` is issued. The destruction of a tuple-space requires no special operation, it is achieved by `in(<?ts>)`.

Tuple-fields can also be of type `ts`. Tuple-spaces on the same level of the hierarchy can be distinguished by adding additional fields as in `eval(<"Q",tsc(Q)>);` and `eval(<"R",tsc(R)>)`. Performing `in(<"Q",?q>)` then retrieves the first one.

[Gelernter, 89] gives examples and points out that `tsc` can be attributed, making tuple-spaces persistent for example. As a consequence, files from an operating system can be represented by a persistent tuple-space. Furthermore, by `in`-ing a tuple-space with active tuples and putting it into a persistent tuple-space with `out`, running programs and file systems are unified.

2.4 Discussion

In the previous sections we reviewed LINDA as a coordination language with special emphasis on the communication of data. It will serve as a starting point for the design of two other coordination languages in the following chapters that share some characteristics with LINDA.

The following list identifies characteristic concepts of LINDA that we take as key-issues for solutions of the coordination problem in any of the aspects we defined in chapter 1.

- **Uncoupling of agents** The basic paradigm of communication is uncoupled in that the sending and receiving agents do not know about each other. This mechanism therefore does not introduce additional concepts on the identification of agents. It is more abstract as the directed communication paradigm, which can well be expressed, as demonstrated in papers referenced below.

- **Associative addressing** An agent willing to receive data uses a pattern or template to address is associatively. It therefore does specify, *what data* it is interested in, not *what message* is wants to receive. The template makes a semantic statement, whereas "deliver message #1012 to me" is a syntactic statement. Again, this mechanism is abstract, as the syntactic identification can well be encoded in a template.

- **Nondeterminism** Associative addressing by templates is non-deterministic, as it does not prescribe the choice of which data to select. During execution the choice finally has to be made, as concrete coordination has to be deterministic. However, the necessary choice is left to some mechanism "behind the stages". This late decision is appropriate in a general view on coordination in which dynamic information should guide decisions. Again, the prescription of the choice can be encoded within the addressing, such as by introducing unique identifiers as fields in a template.

- **Concurrency** Agents being coordinated in a system by LINDA perform their work implicitly concurrently. There are no assumptions implied in what order they execute computation or when they communicate. The only requirement is induced by the potential blocking of `in`/`rd`: Data must be sent by some agent before it can be received.

- **Separation of concerns** LINDA was of the first languages to focus on coordination solely. It demonstrates that this separation of concerns leads to a solution of a coordination problem independent of how computation is performed. The benefits of this separation are concentration on a single problem and abstraction from the solution of other problems such as computation. LINDA's authors make the claim that thereby goals such as simplicity and generality in language-design are matched.

2.5 Bibliographic remarks

LINDA has initiated a variety of research activities. In this section we review the projects at Yale University as well as a number of reported work at other sites.

The classification of three main parallel programming styles in [Carriero and Gelernter, 89a] – result-, specialist- and agenda-parallelism – and their relation to LINDA programming styles has led to a LINDA programming environment, called the LINDA Program Builder ([Ahmed and Gelernter, 91a], [Ahmed and Gelernter, 91b]).

Here, programming with LINDA is supported by templates for the main programming styles, by assisting functions and by macros for combined LINDA-operators. It is implemented as an enhanced version of the Epoch-editor ([Kaplan and Love et al, 92]). The programmer selects a template suitable for his or her problem and fills in – assisted by the editor – the missing parts.

The editor cross-references the complete program text and thus is able to assist the user, e.g. by looking up the use of LINDA operations. Combinations of LINDA operations – e.g. `or-in`, which takes two templates and reads in a tuple matching to one of them – are automatically generated by combining macros and the generation of supplementary tuples.

We do not detail out these mechanisms; they involve mainly syntactic analysis and rely on the accessibility of the complete program in source. Thus the LINDA Program Builder makes an assumption that cannot be up-held in open distributed systems and is outside our focus.

The work on LINDA at Yale University has included investigations in the possibility of optimizations during compile-time by, among others, detecting static access patterns and encoding constant tuples. Strategies are reported in [Bjornson and Carriero et al, 88], [Carriero and Gelernter, 89c], and [Carriero and Gelernter, 91].

Another project at Yale focusses on the use of idle time of workstations in a LAN ([Gelernter and Philbin, 90], [Mattson and Bjornson et al, 92]). The underlying model is that a group of *"piranhas"* is working on a "cloud of tasks". Each piranha tries to grab a piece of work and to execute it. The model is realized by piranha-processes on the workstations in a LAN. An `eval`-operation does not create a process directly but puts it into the cloud of tasks. A *feeder*-process is responsible to initialize, coordinate and finalize their execution by some piranha-process on some workstation in the network. When a workstation has something else to do – e.g. interaction with the user –, the piranha issues a *retreat*-function, which causes the task to be abandoned and rescheduled by the feeder to some other workstation. Thus a distributed version of LINDA makes efficient use of idle time by otherwise unused workstations. Again, such an approach is not applicable to open systems, as the assumption of a LAN in which processes can be rescheduled cannot be upheld in the light of the potential world-wide scale connected by a variety of networks and consisting of various machine-architectures.

Along the projects at Yale University was the design of a parallel LINDA machine ([Ahuja and Carriero et al, 88], [Krishnaswamy and Ahuja et al, 88]). Here, a set of processors equipped with local memory is attached a LINDA-coprocessor that executes the LINDA-primitives in hardware using a separate tuple-store. The processors are laid out in a two-dimensional grid, connected by a set of in- and out-busses. The coprocessor

issues a broadcast on its out-bus for a tuple `out`ed, so that all processors on this bus are supplied with replicas of the tuple. An `in`-request is issued on the in-bus of the processor, which is answered by those processors that find a matching tuple in their tuple-stores. As an in-bus intersects all out-busses, it provides access to the union of all local tuple-stores containing replicas. A protocol ensures that only one tuple is selected for withdrawal and that no two processors withdraw the same tuple. The grid-structure of the in- and out-busses allow for parallel placement and retrieval of tuples.

The claims of the LINDA-designers that LINDA includes concepts that are part of other parallel programming styles but is more general, elegant and more easier than e.g. Concurrent Objects and Actors, Concurrent Logic Programming or functional programming have been laid out in [Carriero and Gelernter, 89b]. The advocates of the named programming models gave replies in [Various authors, 89] which were again answered by the programmatic discussion of coordination languages in [Gelernter and Carriero, 92].

We summarize the confrontation of LINDA and concurrent logic programming here, as is shows some important programmatic claims. In [Various authors, 89], Ehud Shapiro argues that LINDA is Prolog without logic but including concurrency. In his view, the tuple-space could be represented by a multiset of unit-clauses in the Prolog database of facts. `in` and `out` could then be mapped to the *assert* and *retract* mechanisms, where he views matching as a degenerated form of unification.

Whereas LINDA's tuple-space operations are concurrent by definition with a blocking `in`, sequential Prolog does not know about parallelism – which makes the *retract* failing instead of blocking. He gives an executable specification of the LINDA operators by a set of Flat Concurrent Prolog (FCP) clauses that implement `in`, `rd` and `out`.

Shapiro concludes that LINDA-functionality is enclosed in concurrent Prolog and can be made available explicitly by some Prolog-clauses. He therefore argues that concurrent logic programming is a general and versatile programming model that has more expressiveness while retaining efficiency.

Carriero and Gelernter answer in [Carriero and Gelernter, 89b] and [Gelernter and Carriero, 92] that LINDA is more practical and more elegant for parallel computing than Concurrent Logic Programming. They give examples to demonstrate that LINDA-code is easier to understand compared to Concurrent Logic programs. They claim that the latter tends to force the programmer to produce complex solutions to simple problems and suspect this to be caused by selecting a wrong abstraction level and making built-in primitives for parallelism too predetermined for a certain programming style.

They see a difference between – e.g. – FCP and LINDA in the fact that FCP is a complete language whereas LINDA just focusses on the separated aspect of coordination. They argue that introducing several FCP-variants cannot strengthen confidence in the generality of these variants w.r.t. unanticipated problems.

Their concept of separating computation from coordination and embedding LINDA in a host language should make it easier for programmers to take the transition from a sequential to a parallel language. So they would supply him or her with a Prolog-embedding of LINDA instead of introducing a new, parallel Prolog-variant.

Overviews on LINDA applications at Yale University can be found in [Carriero and Gelernter, 88] and [Hupfer and Kaminsky et al, 91]. A vision of future information

systems based on the LINDA coordination model is laid out in [Gelernter, 91] which also covers a set of application classes.

The latest developments at Yale University at the time of writing include the introduction of the coordination language Bauhaus ([Carriero and Gelernter et al, 94]). It has three main differences from LINDA . First, tuples and tuple-spaces are unified by the structure multiset so that multiple tuple-spaces do not need the distinguished data-type `ts` any longer. As a consequence, `in` and `rd` return multisets instead of tuples. Finally, matching is guided by the notion of set-inclusion.

Given the multiset `{a, a, {x, y}}`, `in{x}` returns a multiset that includes all elements of the set `{x}`, which is `{x, y}` in this case. The referenced paper gives examples and discusses Bauhaus' semantics. However, at the time of writing, no further comments can be made on the future direction of the development of Bauhaus.

LINDA has inspired a number of research projects at other sites. A number of embeddings in well-known programming languages have been reported, among these are: [Borrmann and Herdieckerhoff, 88] for Modula-2, [Kane, 91], [Schoinas, 91a], [Schoinas, 91b] for C, [Leler, 90], [Ciancarini and Guerrini, 93] for embeddings at the operating system level accessible in C, [Callsen and Cheng et al, 91] for C++, [Jellinghaus, 90] for Eiffel, [Pinakis, 91] for a Pascal-dialect, and [Sutcliffe and Pinakis, 90], [Sutcliffe and Pinakis, 91] for Prolog-dialects.

Object-oriented embeddings are reported in [Polze, 93], [Polze, 94] for C++ and in [Matsuoka and Kawai, 88], [Matsuoka, 88] for Smalltalk-80. Put shortly, the matching-relation is extended to obey inheritance structures, that is if the type of a tuple-field inherits from a type in a template-field, then both match. This assumes that inheritance induces subtyping between objects from super- and sub-classes.

Reports on embeddings in more research-oriented languages include [Hasselbring, 91] for SETL/E, ELLIS, an embedding in EULISP ([Broadbery and Playford, 91], [Padget and Broadbery et al, 91]), Lucinda, a combination with Russel ([Butcher, 91], [Butcher and Zedan, 91b], [Butcher and Zedan, 91a]) and the parallel programming language *Ease* ([Zenith, 91b], [Zenith, 91a]).

At least two research projects deal with fault-tolerance in a LINDA-like setting. [Anderson and Shasha, 91] describes a LINDA-system called PLinda which introduces primitives for transactions. A process can be made transactional by the primitive xeval. Finer grained transactional activity can be bracketed with xstart–xcommit or –xabort. PLinda also investigates in combined operations like `in-out(tuple)` and extended matching by allowing ranges of values as template-fields.

[Bakken and Schlichting, 91], [Bakken and Schlichting, 93] describe the fault-tolerant FT-LINDA. Here, agents have to follow a fixed protocol of `in`-process-`out`. The tuple-space keeps a backup-copy of the tuple given to an agent. If it fails before completing its work, this copy is given to another agent to process it.

Commercial versions have been reported to be marketed by Scientific Research Associates, New Haven, CT ([Bjornson, 87], [Berndt, 89], [SRA]), LRW Systems, Stamfort, CT for VAX running under VMS ([LRW90], [LRW91b], [LRW91a]), Chorus Supercomputer Inc. ([Cho89]) and Cogent ([Cog89a], [Cog89b], [Cog90]).

The use of LINDA-like concepts in the light of software-engineering and coordination of software development has been studied in [Ciancarini, 93] and [Hasselbring, 93a], [Hasselbring, 93c].

LINDA-matching has been married with Actors ([Agha and Callsen, 92], [Agha and Callsen, 93]). The ActorSpace approach uses associative addressing and uncoupled communication for actors. Here, a matching function is applied to realize a one-to-one-out-of-many communication style, where the address of the actor to which a message is directed is given as a template containing a regular expression. One actor whose address matches the template is chosen non-deterministically for the message delivery.

In [Various authors, 89], Kenneth Kahn and Mark Miller give a historical footnote which refers to the language ETHER whose operations are resembling LINDA. ETHER ([Kornfeld, 79]) is intended to construct problem solving systems which allow for concurrent activities. It does so by having a program being described by a set of rules, called *sprites*, that consist of a pattern and a body. A sprite watches for an assertion to be broadcasted that matches the given pattern. The body may generate new sprites or broadcast assertions.

For a problem solver, the pattern describes a goal that can be evaluated by the sprite. The assertions broadcasted correspond to subgoals to be proved by other sprites. The resulting assertions are collected by giving patterns. A sprite finally broadcasts an assertion signaling that the goal could be proved. ETHER's `when` construct corresponds to LINDA's `in`, and the `broadcast` can be mapped to `out`. Moreover, ETHER knows so called *platforms* that resemble local tuple-spaces. ETHER's approach is also similar to what we have laid out in [Mahr and Tolksdorf, 93].

In this chapter we discussed the communication aspect of coordination by reviewing the coordination Linda. We presented the tuple-space abstraction and the associated operations and illustrated LINDA by examples. After describing tuple-space predicates and multiple tuple-spaces we discussed the characteristics of LINDA's approach to coordination. In the next chapter we focus on the aspects of the synchronization of processes. We do so by introducing a language called ALICE, which – opposed to LINDA – includes its own execution model for processes.

Chapter 3

Coordination focussing on process synchronization: Alice

Referring to our model of coordination in chapter 1, three aspects are of interest: communication of data, synchronization of processes and the coordination of services. In this chapter, we focus on the synchronization aspect. We do so by designing the coordination language Alice which is similar to Linda, but puts more emphasis on processes and their synchronization.

Whereas in Linda `in`, `rd`, `out` and `eval` are embedded into some computation language, Alice takes the role of a host-language for a computation language. Thereby, a model of execution is contained within Alice and is not borrowed from a computation language. How processes are started and synchronized is the main focus of the design of Alice.

Within the course of this chapter we will show how the coordination needs of statements from a simple imperative programming language can be taken care of by Alice. We will define Alice-agents that coordinate loops or conditional constructs.

We thereby learn more about the separation of coordination and computation. In the agent-world in chapter 1, the spectrum of actions ranges from pure computational actions to those we identified as coordination actions. For a coordination language, this spectrum is divided in two parts: those actions covered by a sequential programming language and those covered by a coordination language. Where this division is placed is a design issue.

Linda chooses the point of separation at the coordination end of the spectrum of actions. The design of Alice will show that it also can be chosen so that – for example

– actions concerning the control flow in programs can be covered by a coordination language. To the extreme – as we will see – ALICE even can perform computations.

In ALICE[1] data and processes are represented uniformly as tuples residing in a multiset of tuples called the *agent-space*. If a tuple has only one field and if this one contains a process definition only, this process definition is executed as an agent. The operations in a process definition are coordination operations using the agent-space or computation formulated in some computation language.

The processes which ALICE synchronizes are sequential processes without further control structures. The level of detail on which ALICE coordinates processes is that of basic-blocks.

[Aho and Sethi et al, 86] define a basic block as a sequence of consecutive operations in which one flow of control enters at the beginning and leaves at the end of the block. There is no halt operation within a basic block nor any branch-operations. We consider the execution of a basic as the atomic form of a sequential process. Coordination then deals with actions that start and synchronize basic blocks.

We will show how the arising synchronization problem – e.g. how to start a process that is the else-branch of an if-then-else construct – can be solved by synchronization via the agent-space and its operations only.

ALICE should be understood as an assembler for coordination. It works at a very fine grained level of coordination and does not rule out erroneous programs that run into a deadlock or the like. However, its controlled usage allows for a very high flexibility. ALICE can be taken as an implementation language for other coordination languages as we will show in appendix A. In this chapter we will show how ALICE could be the target-language for a compiler for a simple imperative programming language. In both cases, the concurrent execution of processes and their synchronization is the basic notion of an ALICE-system.

3.1 Tuples and fields in ALICE

The agent-space is populated by tuples, written $\langle \text{field}_0, \ldots, \text{field}_n \rangle$. They are sequences of tuple-fields which are values, tuples or process-definitions. In this subsection we discuss how tuples and fields are constructed. An example of a tuple with a number field, a tuple consisting of a boolean and a character and a boolean value is $\langle 10, \langle \text{T}, \texttt{'a'} \rangle, \text{F} \rangle$.

Values known in ALICE are of the types boolean, character or number. Boolean consists of the truth value – written T – and the false value – written F. Of type character are alpha-numeric symbols such as 'a'...'z'. Values of type number are real number values. All types include a unique bottom value written $\perp_{\text{boolean}}$, $\perp_{\text{character}}$ and $\perp_{\text{number}}$. In addition, there is a type Simple which contains the union of all values of type boolean, character and number and includes a unique value $\perp_{\text{Simple}}$.

[1] This name is taken – how should it be otherwise – from Lewis Carrolls books ALICE'S ADVENTURES UNDERGROUND and THROUGH THE LOOKING GLASS. But to be honest: It is an anagram of the name of the authors cat – CELIA!

ALICE's type- and value-system is completed by the type Tuple containing all tuple-values with a unique element $\bot_{\text{Tuple}}$, and the type process containing all values of process definitions as well as $\bot_{\text{Process}}$, the unique bottom-element of type process.

All of these values can be used as fields in tuples. Compared to LINDA, the bottom elements correspond to formals and there is a richer set of field-types. Besides of the constructor $\langle\rangle$, ALICE has more operations on fields and tuples, these are *expansion*, *templation* and *spreading*.

- *Expansion* means expanding a tuple into its fields and takes a tuple and results in a list of fields. It is therefore an inverse operation to tuple-construction. The expression $\langle\!|\langle \text{field}_0,\ldots,\text{field}_n\rangle|\!\rangle$ results in the fields $\text{field}_0,\ldots,\text{field}_n$. Expansion allows it to denote tuples in a number of ways, if the tuple-constructor is applied to a list of fields resulting from the expansion. For the examples $s=\langle 10,20,30\rangle$ and $t=\langle\bot_{\text{boolean}}\rangle$, the following table shows some expansions:

Tuple s with one field appended	$\langle\langle\!\|s\|\!\rangle,\bot_{\text{character}}\rangle$	$\langle 10,20,30,\bot_{\text{character}}\rangle$
Tuple s with tuple t appended	$\langle\langle\!\|s\|\!\rangle,\langle\!\|t\|\!\rangle\rangle$	$\langle 10,20,30,\bot_{\text{boolean}}\rangle$

  Compared to LINDA, expansion takes the role of decomposing a tuple into its fields but without an explicit notion of a binding rule to an environment such as program variables.

  Expansion has two variants, head-expansion – written $\langle\!|\text{tuple}\langle\!|$ – and tail-expansion – written $|\!\rangle\text{tuple}|\!\rangle$ – which result in the first field of a tuple or the fields of a tuple but the first, resp. Again, these operations allow for flexible ways of constructing tuples:

First field of s between two copies of t	$\langle\langle\!\|t\|\!\rangle,\langle\!\|s\langle\!\|,\langle\!\|t\|\!\rangle\rangle$	$\langle\bot_{\text{boolean}},10,\bot_{\text{boolean}}\rangle$
First and last fields of s surrounding t	$\langle\langle\!\|s\langle\!\|,\langle\!\|t\|\!\rangle,|\!\rangle s|\!\rangle\rangle$	$\langle 10,\bot_{\text{boolean}},20,30\rangle$

- *Templation* is another constructor for tuples: For a field and a tuple it results in a tuple containing as many fields as the given tuple which have the value as given by the field. It is written [field|tuple] and examples are given in the following table:

A tuple that has as many fields as tuple s, each containing an empty tuple	$[\langle\rangle\|s]$	$\langle\langle\rangle,\langle\rangle,\langle\rangle\rangle$
A tuple containing two copies of t	$[\langle\!\|t\|\!\rangle\|\langle\bot_{\text{Tuple}},\bot_{\text{Tuple}}\rangle]$	$\langle\bot_{\text{boolean}},\bot_{\text{boolean}}\rangle$

  Templation is a dynamic tuple-constructor, as the number of fields contained in the tuple is determined at execution time.

- Finally, a set of tuples can be denoted by *spreading* the fields of a tuple. The resulting tuples each contain one field of the tuple. For the example s above, $(\!|s|\!)$ denotes the set of tuples $\langle 10\rangle$, $\langle 20\rangle$ and $\langle 30\rangle$.

  Spreading is used mainly to spread the fields of a tuple into a pattern of tuple-fields. When writing $(\!|1,s|\!)$, the set of tuples $\langle 1,10\rangle$, $\langle 1,20\rangle$ and $\langle 1,30\rangle$ results. The tuple spread – s in the example – may appear only in one location of the pattern. When multiple tuples are mentioned in a spreading, ambiguities have to be resolved by explicitly stating which tuple is to be spread. Thus the expression $(\!|t,s|\!)_{\{s\}}$ results in the set of tuples $\langle\langle\perp_{\text{boolean}}\rangle,10\rangle$, $\langle\langle\perp_{\text{boolean}}\rangle,20\rangle$ and $\langle\langle\perp_{\text{boolean}}\rangle,30\rangle$, whereas $(\!|t,s|\!)_{\{t\}}$ gives $\langle\langle 10,20,30\rangle,\perp_{\text{boolean}}\rangle$. Note that in order to denote $\langle 10,20,30,\perp_{\text{boolean}}\rangle$, one would have to write $(\!|\langle\!|s|\!\rangle,t|\!)_{\{t\}}$ (which is equivalent to $\langle\langle\!|s|\!\rangle,\langle\!|t|\!\rangle\rangle$).

To summarize the operations on tuples and fields: The $\langle\rangle$-constructor results in a tuple whose length and fields are known in advance. Templation with [field|tuple] constructs a tuple whose fields are known and whose number of occurrences depends on the tuple given. Spreading results in a set of tuples whose number and contents depends on the tuple given as argument. The projections $\langle\!|$tuple$|\!\rangle$, $\langle\!|$tuple and $|\!\rangle$tuple$|\!\rangle$ result in the fields of a tuple, in the first field or in all but the first field of a tuple.

We claim that the operations defined are simple and well implementable. A closer examination shows that they are similar to LINDA's handling of fields and tuples. Here, individual fields are accessed by binding their value to some program variable. Also, the tuple constructors, such as spreading and templation, involve only slightly more complexity as the usual tuple-construction with $\langle\rangle$.

So far, we dealt with tuples whose fields were of simple types. We now introduce how process definitions are handled in ALICE as fields and tuples.

3.2 Agents in ALICE

As stated, tuple fields in ALICE can contain a process definition. When a tuple contains a process definition as its only field, its operations are executed sequentially in which case we speak of an *agent*.

Processes are composed of operations that emit or withdraw tuples from the agent-space or perform local computation. out(tuple) emits a tuple to the agent-space from which it can be withdraw by agents. in(tuple) searches the agent-space for a tuple that *matches* the tuple given as the argument. If it does not find one, the operation blocks until another agent out's a matching tuple. local denotes computations expressed in some computation language that is embedded into ALICE. They are only required to conform to the execution model of ALICE, that is that local is started immediately and that it has a defined end at which the next operation in the process definition can be started. Thus local may well involve concurrent computations, however, it must not start some thread of control that continues to run when the next operation from the process definition is started.

Two tuples are said to be matching if they have the same length, if the types of the tuple-fields are pairwise the same and the values of the fields match. Matching of values

is defined as equality for ordinary values, whereas the bottom element of a type matches any value of the type, including the bottom element itself. Figure 3.1 formalizes the matching relation, where fields are written as pairs $a \in \tau$ with a being a value of type τ. The type of $\bot_\tau$ is τ.

$$\begin{aligned}
\text{match}(a \in \tau, a \in \tau) &= \text{TRUE} \\
\text{match}(a \in \tau, \bot_\tau) &= \text{TRUE} \\
\text{match}(\bot_\tau, a \in \tau) &= \text{TRUE} \\
\text{match}(a \in \text{Tuple}, b \in \text{Tuple}) &\text{ if } \textstyle\bigwedge_{i=1}^{n} : \text{match}(a_i, b_i) \\
\text{match}(a \in \text{Process}, b \in \text{Process}) &\Leftrightarrow a \sim b
\end{aligned}$$

Figure 3.1: The matching relation

Identical fields of the same type match in any case as well as the bottom element of a type matches any value of that type. For two values of type tuple, their length has to be identical with pairwise matching fields. For process definitions as described, we introduce an abstract matching based on some equivalence relation $\sim$. This relation could be syntactic equivalence or some decidable semantical equivalence. In the following we will use only the matching between $\bot_{\text{Process}}$ and process definitions.

ALICE's matching relation differs from the one defined for LINDA in that the bottom elements are treated as values and that bottom elements of the same type match. In LINDA, formals can appear in templates only and are not included in the matching relation. ALICE's definition has a specific consequence: When performing an in($\langle \bot_{\text{number}} \rangle$), for example, the tuple retrieved can contain a number value or the bottom element.

In most LINDA-like languages this is considered impractical as it is unclear how the bottom element should be interpreted by a computational language. This argument, however, does not hold if the retrieved tuple is not interpreted by a computational language but just used for further coordination-operations. Also, it does not hold if one knows if the agent-space from which the tuple is retrieved contains such elements. In subsection 3.2.1 we will introduce means for an agent to establish a local agent-space which is filled with tuples exclusively by this agent and for which this definition of the matching-relation proves to be useful.

Agents in ALICE consist of a process and a set of data, being a set of fields (remember that tuples are treated as fields), called the *local environment*. Processes reference this data by names, resulting in a binding of a field to a name and the access to a field by a name. The naming scheme is purely local to the agent and names cannot be referenced from another agent.

An example of an agent is $\langle$in(a:$\bot_{\text{Tuple}}$).out(a).out(a)$\rangle$, which retrieves a tuple from the agent-space and binds it to the name a in its local environment. It references it by the out-operations twice. Let a tuple consisting of a number-field and a process-definition exist in the agent-space as $\langle$10,in(a:$\bot_{\text{Tuple}}$).out(a).out(a)$\rangle$. Then, some agent can retrieve it by executing in(a:$\langle \bot_{\text{number}}, \bot_{\text{Process}} \rangle$), after which $\langle$a$\langle$ references the field 10 and $\rangle$a$\rangle$ references the field in(a:$\bot_{\text{Tuple}}$).out(a).out(a). However, the two references to a in the agent executed and the process-definition retrieved are completely unrelated.

As stated above, the creation of an agent is performed by some agent by out-ing a tuple with a single field containing a process definition. Executing out($\langle$in(a:$\perp_{\text{Tuple}}$). out(a).out(a))$\rangle$) results in a new agent that tries to retrieve a tuple from the agent-space and emits it twice.

When this agent is executed, it consists of a set of data – the local environment –, a process that has been executed already and a process that will be executed. We use the notation $\langle\!\langle$\{data\},executed,to_execute$\rangle\!\rangle$ for looking at an agent in this structure. If p is a process definition, then the corresponding agent can be written $\langle\!\langle\{\},,p\rangle\!\rangle$ when its execution starts. For the example above, the execution of the agent looks as follows:

After the initialization	$\langle\!\langle$\{a\},,in(a:$\perp_{\text{Tuple}}$).out(a).out(a)$\rangle\!\rangle$
After retrieving a tuple $\langle 10\rangle$	$\langle\!\langle$\{a:$\langle 10\rangle$\},in(a:$\perp_{\text{Tuple}}$),out(a).out(a)$\rangle\!\rangle$
After emitting a twice	$\langle\!\langle$\{a:$\langle 10\rangle$\},in(a:$\perp_{\text{Tuple}}$).out(a).out(a),$\rangle\!\rangle$

The local environment is a mapping from identifiers to values. It is modified by in as in the example and by any local computation. It is accessed by out and any local computation. An embedding of a computation language into ALICE provides the local-operations and has to implement some mechanisms that makes access to the local environment possible.

When writing an ALICE-program, one defines agents by giving them a name, listing the identifiers used in the local environment and by a process definition to be executed. An example is:

$$\text{in-twiceout:} \quad \langle\!\langle\{a\},,\text{in(a:}\perp_{\text{Tuple}}\text{).out(a).out(a)}\rangle\!\rangle$$

Note that the name is used only in the program text and does not exist at runtime. Given such a definition, another agent can use it as follows:

$$\text{out-twice-user:} \quad \langle\!\langle\{\},,\text{out}(\langle 10\rangle).\text{out(in-twiceout)}\rangle\!\rangle$$

There is also a notation that initializes the local environment when performing an out with an agent. Let twice-out be an agent defined as

$$\text{twice-out:} \quad \langle\!\langle\{a\},,\text{out(a).out(a)}\rangle\!\rangle$$

Then another agent can create twice-out as an agent and initialize its local environment with the tuple $\langle 10\rangle$ if it is defined as

$$\text{twice-user:} \quad \langle\!\langle\{\},,\text{out}(\langle\text{twice-out}\rangle_{\{\langle 10\rangle\}})\rangle\!\rangle$$

In this notation, a list of fields is given in the initialization list which is bound as values to the identifiers used in the local environment in the order they are defined[2]. The execution of twice-user results in the agent $\langle\!\langle$\{a:$\langle 10\rangle$\},,out(a).out(a)$\rangle\!\rangle$.

Spreading as defined above takes a tuple and a field-pattern and spreads the fields of the tuple resulting in a set of tuples. It also can be applied to agent-tuples so that a set of agents is generated. Again, all references in the pattern are replaced with a tuple containing one field from the spread tuple. If there is no reference to tuples in the process, no replacement takes place in the replicated agents and a set of identical agents is denoted. Given that s=$\langle$10,20,30$\rangle$, $(\!|$twice-out$|\!)_{\{s\}}$ denotes the following set of agents:

[2]There is no possibility of reference from outside to the names used in an agents.

$$\langle\!\langle \{a{:}\langle 10\rangle\},,\text{out(a).out(a)}\rangle\!\rangle$$
$$\langle\!\langle \{a{:}\langle 20\rangle\},,\text{out(a).out(a)}\rangle\!\rangle$$
$$\langle\!\langle \{a{:}\langle 30\rangle\},,\text{out(a).out(a)}\rangle\!\rangle$$

As for ALICE's operations on fields and tuples, we claim that the operations such as initialization and spreading of agents are well implementable. Given that the local environment can be represented as an array of fields, initialization means simple copy-operations amongst fields. The same argument applies to the spreading of tuples over agents. Access to the local environment can easily be compiled into accesses to the array of fields. The concept of dividing the process definition in a part that has been executed and a part that will be executed corresponds to an implementation using some program counter.

To summarize: Process-definitions are composed of in, out and local, where in retrieves a matching tuple from the agent-space, out emits one and local are operations from a computation language. A tuple consisting of a process field only is executed as an agent.

Agents consist of a local environment, a process that has been executed and a process that will be executed. The local environment can be initialized during an out with tuples. Spreading can be applied to agents as well and results in a set of agents whose local environment is initialized with fields from a tuple.

3.2.1 Local agent-spaces

The out- and in-operations emit and withdraw tuples to and from the agent-space in which the agent works. ALICE introduces the notion of *local agent-spaces* which can be set up locally by an agent and are initialized with tuples and agents. The agents therein are executed and manipulate the local agent-space. Executions in the local agent-space end when all agents either have terminated or block at an in-operation. This final state is taken as the result of a local agent-space as a value of type boolean. The programmer has to assure that these final states are reached[3].

Local agent-spaces are defined in ALICE as a variant of the in-operation. The value retrieved is the result of the local agent-space as a tuple containing a boolean-field. It is notated as in(name:{tuple_0,...,tuple_n,agent-tuple_0,...,agent-tuple_m}).

As an example, a local agent-space can be used to test if two tuples match without accessing the global agent-space and without blocking:

$$\langle \ldots \text{in(c:\{a,}\langle\text{in(b)}\rangle\text{\})}\ldots\rangle$$

Here, a local agent-space is initialized with the tuple a and the agent-tuple in(b). Execution of that agent is started and this execution can result in a termination of all agents when a matches b, or in a blocking of all agents in the case that a does not match b. This result of the local agent-space then is bound to the name c in the local environment as tuple $\langle F\rangle$ or $\langle T\rangle$.

Spreading is useful to initialize a local agent-space with sets of tuples and agents. To test if tuple a contains at least the fields from tuple b, a process can execute

[3]Recall that ALICE is a "coordination assembler"!

$$\langle \ldots \text{in}(\text{c:}\{(\!|\text{a}|\!),(\!|\text{in}(\langle \text{b} \rangle)|\!)\}) \ldots \rangle$$

Here, the tuple a contains at least the fields of b if the local agent-space terminates. For tuples a=⟨10,20,30⟩ and b=⟨30,10⟩ the spreading initializes the agent-space with ⟨10⟩, ⟨20⟩, ⟨30⟩ and the agents ⟨in(⟨30⟩)⟩ and ⟨in(⟨10⟩)⟩. A similar test can be performed by combining templation and a local agent-space by performing

$$\langle \ldots \text{in}(\text{c:}\{\text{a}, \langle \text{in}([\bot_{\text{number}}|\text{a}])\rangle\}) \ldots \rangle$$

This tests if all fields of a are of type number. For a=⟨10,20,30⟩ the local agent-space is initialized with the tuple ⟨10,20,30⟩ and the agent in(⟨$\bot_{\text{number}}$,$\bot_{\text{number}}\bot_{\text{number}}$⟩). The result of the agent-space here is T and would be F for a=⟨10,'A',30⟩.

If we wanted to know if a has at least as many boolean fields as b has fields, spreading without a reference can be applied:

$$\langle \ldots \text{in}(\text{c:}\{(\!|\text{a}|\!),(\!|\text{in}(\langle \bot_{\text{boolean}} \rangle)|\!)_{\{\text{b}\}}\}) \ldots \rangle$$

For a=⟨10,T,30,F⟩ and b=⟨'A',30⟩, the agent-space is initialized with ⟨10⟩, ⟨T⟩, ⟨30⟩, ⟨F⟩, ⟨in(⟨$\bot_{\text{boolean}}$⟩)⟩ and ⟨in(⟨$\bot_{\text{boolean}}$⟩)⟩, in which case the test succeeds.

To formalize the constructs discussed in the preceding sections, figure 3.2 shows the grammar over which process definitions and tuples are generated in ALICE.

Process ::= Operation **.** Process | Operation

Operation ::= **out(** Tuple **)** | **out(** Identifier **)** | **out(** Spreading **)** |
in(Tuple **)** | **in(** Identifier **:** Tuple **)** | **in(** Identifier **: {** Tuples**}** **)** |
local

Tuples ::= Tuple | Tuple **,** Tuples | Spreading

Spreading ::= (|Fields|) | $(\!|\text{Fields}|\!)_{\{\text{Tuple}\}}$

Tuple ::= Identifier | ⟨ Fields ⟩ | [Field|Tuple] ⟪ **{** Tuples **},** Process **,** Process ⟫

Fields ::= Field **,** Fields | Field | ⟨|Tuple⟩ | |⟩Tuple⟩

Field ::= Identifier | $\bot_{\textbf{Tuple}}$ | Tuple | ⟨|Tuple⟨ |
$\bot_{\textbf{Simple}}$ | Number | $\bot_{\textbf{number}}$ | Character | $\bot_{\textbf{character}}$ | Boolean | $\bot_{\textbf{boolean}}$
|
$\bot_{\textbf{Process}}$ | Process | Process **{** Tuples **}**

Figure 3.2: Grammar for ALICE

3.3 Process synchronization with ALICE

In the introduction to this chapter, we discussed coordination of processes at a very fine grained level. Processes in ALICE are sequentially executed from the first operation to the last without any forms of control flow constructs. ALICE-processes can be understood as *basic blocks* following the definition in [Aho and Sethi et al, 86].

A basic block is a minimal unit in the execution of programs. The synchronization aspect of coordination is concerned with actions for the start, blocking, unblocking and ending of processes as defined in our model in chapter 1. Thus, any form of control constructs such as loops etc. requires coordination.

In a sequential environment this synchronization aspect is not prominently visible and is implemented by branches in the program code. However, when changing to a parallel or distributed environment, it becomes well noticeable that a real coordination problem has to be solved.

How a solution to coordinate the synchronization of basic blocks can guide the design of a multi-computer has been demonstrated in [Dai, 88], [Dai and Giloi, 90a], [Dai and Giloi, 90b]. Here, a machine architecture involving two separate levels of execution has been proposed that uses RISC processors for the sequential execution of basic blocks, and a *graph-level* in which coordination of basic blocks is performed based on dataflow-graphs.

In this section we demonstrate how ALICE can be used to formulate agents that perform coordination as known from imperative programming constructs. As ALICE-agents work in parallel the result is a system in which basic-blocks are executed in parallel.

In a simple imperative programming languages, such as the one defined in [Hennessy, 90], one finds boolean and numerical expressions, assignment, sequencing, if-then-else-clauses and while-do-loops. In ALICE, expressions are abstracted from by local. Sequencing is expressed with ".". Assignment is performed by local or by in(identifier:...). In the next subsections, we deal with if-then-else and while-do.

In the introduction, we classified ALICE as "assembler-like". The examples shown here therefore should be understood as patterns that a compiler emits for coordination actions.

3.3.1 Coordinating conditional execution

In an if-then-else clause, we can identify two processes for which coordination is required: One branch of the clause has to be started as a process according to the condition. In ALICE, such a construct can be represented as a tuple of the form ⟨condition,if-process,else-process⟩.

Starting one process according to the condition is performed by the following two agents:

bool-cond-t:
⟨in(⟨T,a:$\perp_{\text{Process}}$,$\perp_{\text{Process}}$⟩).out(⟨a⟩).out(⟨bool-cond-t⟩)⟩
bool-cond-f:
⟨in(⟨F,$\perp_{\text{Process}}$,a:$\perp_{\text{Process}}$⟩).out(⟨a⟩).out(⟨bool-cond-f⟩)⟩

The two agents initialize the first process as an agent if the condition-field matches the boolean value T, the second otherwise. An if-then-else construct in ALICE then can be executed by an agent by computing the condition with local operations and by performing an out of a tuple of the above form.

3.3.2 Coordinating loops

For the control construct while-do at least three processes of interest can be identified: the computation of the condition for the loop, the loop body and the process to be executed after the termination of the loop.

Let the while-do operate on a local environment given as a set of fields $t_1,\ldots,t_n$. Let the local computation evaluating the loop condition into a boolean field c be called b_1, the one of the loop-body b_2 and the process after the loop b_3. We can formulate ALICE-agents that perform the coordination of this loop.

First, an extended version of the if-then-else construct is defined that passes a local environment to the processes coordinated:

env-cond-t:
$\langle \text{in}(\langle \text{T,a:}\bot_{\text{Process}},\bot_{\text{Process}},\text{e:}\bot_{\text{Tuple}}\rangle).\text{out}(\langle \text{a}\rangle_{\{\langle\!|\text{e}|\!\rangle\}}).\text{out}(\langle \text{env-cond-t}\rangle)\rangle$
env-cond-f:
$\langle \text{in}(\langle \text{F,}\bot_{\text{Process}},\text{a:}\bot_{\text{Process}},\text{e:}\bot_{\text{Tuple}}\rangle).\text{out}(\langle \text{a}\rangle_{\{\langle\!|\text{e}|\!\rangle\}}).\text{out}(\langle \text{env-cond-f}\rangle)\rangle$

Here, the selected process it turned into an agent initialized with a local environment as found in the if-then-else-tuple. We use the env-cond agents for the coordination of the loop in which two agents are involved. The agent looper coordinates the start of the loop-body or the process after the loop depending on a condition:

$$\text{looper:}\quad \langle\!\langle \{t_1,\ldots,t_n,c\},,b_1.\text{out}(\langle\langle\!|c|\!\rangle,\text{body,continue,}\langle t_1,\ldots,t_n\rangle\rangle)\rangle\!\rangle$$

b_1 is some local computation that evaluates a condition and stores it in the field c as a boolean. Then, a tuple for the env-cond-agents is emitted containing the loop-body to be executed when the condition was true, containing the process after the loop.
The loop-body is executed by the following agent:

$$\text{body:}\quad \langle\!\langle \{t_1,\ldots,t_n\},,b_2.\text{out}(\langle \text{looper}\rangle_{\{t_1,\ldots,t_n\}})\rangle\!\rangle$$

b_2 is the local computation contained in the loop-body. The following out coordinates the next iteration of the loop by emitting the looper agent. Finally, the process to be executed after the loop is formulated as:

$$\text{continue:}\quad \langle\!\langle \{t_1,\ldots,t_n\},,b_3\ldots\rangle\!\rangle$$

The scheme outlined by these agents performs coordination for a sequential loop where the loop-body is assumed to modify the environment so that the evaluation of the loop-condition or the next iteration depends on it. Loops can be suited for parallel execution of their bodies if these data-dependencies do not exist. For ALICE such loops make sense as their bodies can contain out-operations that have effects on other agents. In this case the following par-looper agent can be defined:

par-looper: $\langle\!\langle \{t_1,\ldots,t_n,c\},,b_1.$
$\quad out(\langle c,par\text{-}looper,continue,\langle t_1,\ldots,t_n\rangle\rangle).$
$\quad out(\langle c,b_2,,\langle t_1,\ldots,t_n\rangle\rangle)\rangle\!\rangle$

Here, the next evaluation of the loop condition is generated as an agent as well as an agent executing the loop-body. If the loop is to terminate, the continue agent is emitted and no new loop body executes. Note that the local environment of par-looper is used to initialize b_2 so that there may be no data-dependencies between continue and the loop-body.

3.3.3 Synchronization of groups of agents

If all body-agents have to terminate prior to the execution of continue, an additional synchronization need arises. It can be formulated as an ALICE-agent by extending the loop-agents.

The idea is that such a loop gets a unique label l and introduces a signal-tuple s. The agent synch-looper "counts" the number of body-agents generated in a tuple of the form $\langle l,s\rangle$, where s has the form $\langle\langle\rangle,\ldots,\langle\rangle\rangle$ and the number of empty tuples in s corresponds to the number of bodies emitted. There is one tuple of the form $\langle l,\langle\rangle\rangle$ emitted at the end of the body-generations. Each body retrieves it after the execution of its computations and appends an empty tuple. The continuation then is synchronized by the existence of a tuple $\langle l,\langle\langle\rangle,\ldots,\langle\rangle\rangle\rangle$ which exists only when all body-agents have terminated.

synch-looper as shown below computes the loop-condition with b_1. If it yields true, another synch-looper will be generated with the signal-tuple enlarged by an empty-tuple. Also, one body-agent will be emitted by the env-cond-agents. If the loop is to end, the emit-signal agent will be generated with the number of generated bodies encoded in the signal-tuple.

synch-looper:
$\langle\!\langle \{t_1,\ldots,t_n,l,s,c\},,b_1.$
$\quad out(\langle\langle\!|c|\!\rangle,synch\text{-}looper,,\langle t_1,\ldots,t_n,l,\langle\langle\!|s|\!\rangle,\langle\rangle\rangle\rangle\rangle).$
$\quad out(\langle\langle\!|c|\!\rangle,synch\text{-}body,,\langle t_1,\ldots,t_n,l\rangle\rangle).$
$\quad out(\langle\langle\!|c|\!\rangle,,emit\text{-}signal,\langle t_1,\ldots,t_n,l,s\rangle\rangle).$
$\quad \rangle\!\rangle$

The emit-signal agent – which is executed after all body-agents have been generated – first out's a tuple $\langle l,\langle\rangle\rangle$ as the initial signal-tuple. Also, it starts the synch-continue agent initialized with the final signal-tuple:

emit-signal: $\langle\!\langle \{t_1,\ldots,t_n,l,s\},,out(\langle l,\langle\rangle\rangle).out(\langle synch\text{-}continue\rangle_{\{t_1,\ldots,t_n,\langle l,s\rangle\}})\rangle\!\rangle$

A body-agent first performs its computation with b_2 and then retrieves the signal-tuple with the label l. The second part of this tuple containing the number of terminated body-agents encoded as empty tuples then is passed to the re-emit-signal agent, which appends another empty tuple and emits it to the agent-space.

synch-body: $\langle\!\langle \{t_1,\ldots,t_n,l,m\},,b_2.$
$\quad in(\langle m{:}\langle l,\perp_{\text{Tuple}}\rangle\rangle).$
$\quad out(\langle \text{re-emit-signal}\rangle_{\{l,|m\}})\rangle\!\rangle$

re-emit-signal: $\langle\!\langle \{l,s\},,out(\langle l,\langle\langle s\rangle,\langle\rangle\rangle\rangle)\rangle\!\rangle$

synch-continue finally, waits for the signal-tuple in its final state as it has been initialized by the emit-signal agent. It is present in the agent-space exactly when all body-agents have terminated and the last re-emit-signal has out-ed it.

synch-continue: $\langle\!\langle \{t_1,\ldots,t_n,s\},,in(s).b_3\ldots\rangle\!\rangle$

With these examples we have shown how ALICE-agents perform coordination for the synchronization of processes involved in constructs known from imperative programming. They should be understood as the output of some compiler that uses ALICE as the target assembler language.

The last examples involved the encoding of a counter in a tuple by filling it with empty tuples. We notice that even computation can be performed in ALICE, and will demonstrate this in the next section by defining an interpreter for Turing-machines with ALICE-agents.

3.4 Turing machines with ALICE

Let a Turing-machine be described by a set of states Q, a set of tape symbols Γ, a set of input-symbols Σ, the blank B, the transition function δ, mapping $Q \times \Gamma$ into $Q \times \Gamma \times \{L, R\}$, the initial state q_0 and a set of terminal states F.

Then we can specify ALICE-agents that interpret this Turing machine with respect to an input α_0. We code the tape and the current state in a tuple of four fields. The first contains the current state encoded as $\langle q\rangle$. The second represents the tape to the left from the head in reversed order with a tuple for each field. The third contains the symbol under the head in a tuple and the last represents the tape right from the head.

If we have a Turing machine in state q and a tape $a_1a_2\ldots a_{i-1}a_m a_{i+1}\ldots a_n B\ldots$ our agent would code it as the tuple

$$\langle\langle q\rangle, \langle a_{i-1}, a_2, a_1\rangle, \langle a_i\rangle, \langle a_{i+1}, \ldots, a_n\rangle\rangle$$

With this representation, two agents L and R can be formulated that perform the movement of the machine's head to the left and to the right:

R: $\langle\!\langle \{q,l,x,r\},,out(\langle q,\langle x,\langle|l|\rangle,\langle\langle r|\rangle,\langle|\rangle r|\rangle,\text{' '}\rangle\rangle\rangle)\rangle\!\rangle$
L: $\langle\!\langle \{q,l,x,r\},,out(\langle q,\langle|\rangle l|\rangle,\langle\langle l|\rangle,\langle x,\langle r|\rangle\rangle\rangle\rangle)\rangle\!\rangle$

R performs a movement to the right, thus putting the symbol under the head in the tuple representing the tape to the left, and putting the first symbol from the tape to the right under the head. L does the opposite moving the symbol under the head to the right and putting the symbol to the left under the head.

Let a transition from δ be coded as a tuple with five fields: $\langle$q,x,p,y,a$\rangle$. Here, q is the state of the Turing machine, x is the current symbol on the tape, p is the state after the transition with the current symbol replaced by y. a is an agent that performs the movement of the head, for which we can use agents L and R.

This representation then is used by a set of δ-agents that are initialized with these fields. A δ-agent is defined as

$$\begin{aligned}\delta\text{-agent:}\quad \langle\!\langle \{\text{q,x,p,y,a,l,r}\},,&\text{in}(\langle \text{q,l:}\bot_{\text{Tuple}},\text{x,r:}\bot_{\text{Tuple}}\rangle).\\ &\text{out}(\langle \text{a}\rangle_{\{\text{p,l,y,r}\}}).\\ &\text{out}(\langle \delta\text{-agent}\rangle)\rangle\!\rangle\end{aligned}$$

According to the transition, it does an in for a tape represented by a tuple which contains the matching state q and symbol x under the head. Then, the transition is taken by replacing the state with the new one p and by overwriting the symbol under the head with y. The new tape-representation is the initialization for the movement agent a. Finally, the δ-agent re-emits itself to the agent-space.

Given, we have a transition in state q_0 and input symbol 1, a δ-agent for the transition into state q_3 with the symbol replaced by 0 and a movement to the left, will perform in$(\langle$'0',l:$\bot_{\text{Tuple}}$,'1',r:$\bot_{\text{Tuple}}\rangle)$ to get a matching tape and then out$(\langle \text{L}\rangle_{\{'3',\text{l},'0',\text{r}\}})$. We will encode states as characters representing the number of the state, so that state q_4 is represented by '4'.

A Turing-machine interpreter in Alice is to be initialized with an initial state q0, the terminal state f, the input as a tuple a and the δ-function as a tuple consisting of transition-tuples in the form suited for δ-agents.

$$\begin{aligned}\text{turing:}\quad \langle\!\langle \{\text{q0,f,d,a}\},,&\text{out}(\langle\langle \text{q0}\rangle,\langle\rangle,\langle\langle\!| \text{a} |\!\rangle,\langle |\!\rangle \text{a} \langle\!| \rangle\rangle\rangle).\\ &\text{out}(|\!|\langle \delta\text{-agent}\rangle_{\{|\!| \text{d} |\!|\}}|\!|).\\ &\text{out}(|\!|\langle \text{in}(\langle \text{f,}\bot_{\text{Tuple}},\bot_{\text{Tuple}},\bot_{\text{Tuple}}\rangle)\rangle|\!|))\rangle\!\rangle\end{aligned}$$

turing first emits the initial tape tuple with the initial state, no input to the left of the head, the first field from the input-tuple under the head and the tail as the remaining input on the right of the head. Then a set of δ-agents is generated by spreading the transition-tuples contained in d. Finally, an agent is emitted that absorbs the tape tuple if it represents the Turing machine in a terminal state.

As an example we take a Turing machine that recognizes the language $L = \{0^n 1^n | n \geq 1\}$ as defined in [Hopcroft and Ullman, 79]. It is given by states $Q = \{q_0, q_1, q_2, q_3, q_4\}$, tape-symbols $\Sigma = \{0, 1\}$, input-symbols $\Gamma = \{0, 1, X, Y, B\}$, the initial state q_0 and the final state $f = q_4$. The transition-function δ is given in figure 3.3.
δ is encoded in the following tuple:

d=
$\langle\langle$'0','0','0','X', R$\rangle$, $\langle$'0','Y','3','Y', R$\rangle$,
$\langle$'1','0','1','0', R$\rangle$, $\langle$'1','1','2','Y', L$\rangle$, $\langle$'1','Y','1','Y', R$\rangle$,
$\langle$'2','0','2','0', L$\rangle$, $\langle$'2','X','0','X', R$\rangle$, $\langle$'2','Y','2','Y', L$\rangle$,
$\langle$'3','Y','3','Y', R$\rangle$, $\langle$'3','B','4','B', R$\rangle\rangle$

The interpretation of this machine is started by the agent $\langle \text{turing}\rangle_{\{'0','4',\text{d},\langle'0','0','1','1'\rangle\}}$ on the input 0011. The tape-tuple has the following outforms during its interpretation:

State	Input 0	1	X	Y	B
q_0	(q_0, X, R)	—	—	(q_3, Y, R)	—
q_1	$(q_1, 0, R)$	(q_2, Y, L)	—	(q_1, Y, R)	—
q_2	$(q_2, 0, L)$	—	(q_0, X, R)	(q_2, Y, L)	—
q_3	—	—	—	(q_3, Y, R)	(q_4, B, R)
q_4	—	—	—	—	—

Figure 3.3: δ of a Turing-machine that recognizes 0^n1^n

```
⟨'0',⟨⟩,'0',⟨'0','1','1',' '⟩⟩
⟨'1',⟨'X'⟩,'0',⟨'1','1',' '⟩⟩
⟨'1',⟨'X','0'⟩,'1',⟨'1',' '⟩⟩
⟨'2',⟨'X'⟩,'0',⟨'Y','1',' '⟩⟩
⟨'2',⟨⟩,'X',⟨'0','Y','1',' '⟩⟩
⟨'0',⟨'X'⟩,'0',⟨'Y','1',' '⟩⟩
⟨'1',⟨'X','X'⟩,'Y',⟨'1',' '⟩⟩
⟨'1',⟨'X','X','Y'⟩,'1',⟨' '⟩⟩
⟨'1',⟨'X','X','Y'⟩,'1',⟨' '⟩⟩
⟨'2',⟨'X','X'⟩,'Y',⟨'Y',' '⟩⟩
⟨'2',⟨'X'⟩,⟨'X'⟩,⟨'Y','Y',' '⟩⟩
⟨'0',⟨'X','X'⟩,'Y',⟨'Y',' '⟩⟩
⟨'3',⟨'X','X','Y'⟩,'Y',⟨' '⟩⟩
⟨'3',⟨'X','X','Y','Y'⟩,' ',⟨' '⟩⟩
⟨'4',⟨'X','X','Y','Y',' '⟩,,⟩
```

The final tuple is absorbed by the agent that is emitted by turing depending on the final state.

We have shown that it is possible to write a set of agents in ALICE that interpret a Turing machine. Thereby we have proven that ALICE has the ability to compute every recursive enumerable function.

The interesting point is that our focus in the design of ALICE was coordination. It turns out that the model is applicable to computation also. We take this as a prove for the claim in [Gelernter and Carriero, 92] that coordination and computation are orthogonal concepts.

However, we like to point out that a coordination language is different from a computational language as the objectives of its usage are different. The examples in which ALICE has been used so far have led to very short solutions in terms of the complexity of agents when coordination issues were concerned.

3.5 Historical and bibliographic remarks

The name ALICE has been used for a parallel reduction machine in the early 80's. Despite the pure coincidence of the names, a look at the ALICE-machine shows a

connection between reduction-oriented and LINDA-like generative computation. The ALICE-machine is described in detail in [Darlington and Reeve, 81]; our description is based on the introduction found in [Moor, 82].

In the ALICE-machine, the basic unit of work is called a *packet*. A program is represented as a number of packets that are held in a packet-pool. Processors try to retrieve and evaluate packets from the pool that have been marked as processable. The evaluation comprises of the execution of code and the rewrite of a packet which is re-delivered to the pool. If a packet cannot be evaluated due to the lack of arguments, it is put into the pool, appropriately marked and rewritten with pointers to the argument packets awaited. A successful evaluation can make packets that await arguments executable which subsequently leads to their evaluation by some processor.

The concept of concurrent agents coordinated by using a shared multiset of elements also can be found in the language GAMMA, an acronym for **G**eneral **A**bstract **M**odel for **M**ultiset M**a**nipulation ([Banâtre and Le Métayer, 91], [Banâtre and Le Métayer, 93], [Mussat, 91]). It is motivated by the distinction between logical and physical parallelism. GAMMA is designed as a high-level language in which algorithms are described without introducing artificial sequentiality.

The abstraction introduced in GAMMA is that of a chemical reaction. A multiset is thought of as a chemical solution – its elements are molecules. A GAMMA program defines a *reaction condition* for which molecules for a reaction have to be found and selected and an *action* describing the result of the reaction – a new molecule. The stable state of the solution – i.e. when no reaction condition applies to the solution – is identified with the termination of the program.

More formally, a GAMMA program defines reaction conditions and actions (R,A) that, when applied to a multiset replace a subset of elements $\{x_1,\ldots,x_n\}$ by the elements of $A(x_1,\ldots,x_n)$ if $R(x_1,\ldots,x_n)$ is true. The execution of a GAMMA program means the application of the Γ operator, which controls the evaluation of a set of reaction conditions and actions on a multiset. The Γ-operator provides the abstraction: If several reaction conditions hold on a multiset, one is chosen non-deterministic.

Figure 3.4 shows some examples taken from [Banâtre and Le Métayer, 93][4]. Factorial (3.4(a)) computes n!. The reaction rule R always applies on two elements of a multiset. If such two elements are available, the action A defines a reaction of these two element to their product. Both are applied with the Γ-operator to a multiset in which all numbers from 1 to n are present. The computation terminates if there is only one element left, in which case the element is n!.

Sort (3.4(b)) shows an algorithm for sorting. Here, the elements of the multiset, on which R and A are applied, contain an array represented by pairs (index,value). R states that if two elements of the multiset satisfy the condition that the index of the first is higher than that of the second and its value is lower than the second, the reaction A can be applied, which swaps the values.

Finally, Philosophers (3.4(c)) gives a solution to the dining philosophers problem. The multiset on which the reactions are applied contains representations of the philosophers (P_i) and four forks (F_i). The first reaction rule R_1 states that two adjacent forks

[4] [Mussat, 91] gives more examples and shows how to derive GAMMA-programs from mathematical specifications.

$$\begin{aligned}&\text{fact(n)} = \Gamma((\text{R,A}))\ (\{1,\ldots,\text{n}\})\\&\textbf{where}\\&\qquad \text{R(x,y)} = \text{true}\\&\qquad \text{A(x,y)} = \{\text{x*y}\}\end{aligned}$$

(a) Factorial

$$\begin{aligned}&\text{sort(Array)} = \Gamma((\text{R,A}))\ (\text{Array})\\&\textbf{where}\\&\qquad \text{R((i,v),(j,w))} = (\text{i}>\text{j})\ \textbf{and}\ (\text{v}<\text{w})\\&\qquad \text{A((i,v),(j,w))} = \{(\text{i,w}),(\text{j,v})\}\end{aligned}$$

(b) Sorting

$$\begin{aligned}&\text{philosophers} = \Gamma((\text{R}_1,\text{A}_1)\ (\text{R}_2,\text{A}_2)\ (\{\text{F}_0,\text{F}_1,\text{F}_2,\text{F}_3,\text{F}_4\}))\\&\textbf{where}\\&\qquad \text{R}_1\ (\text{F}_i,\text{F}_{i\oplus l}) = \text{true}\\&\qquad \text{A}_1\ (\text{F}_i,\text{F}_{i\oplus l}) = \{\text{P}_i\}\\&\qquad \text{R}_2\ (\text{P}_i) = \text{true}\\&\qquad \text{R}_2\ (\text{P}_i) = \{\text{F}_i,\text{F}_{i\oplus l}\}\end{aligned}$$

(c) Dining philosophers

Figure 3.4: Examples in GAMMA

are consumed by an eating philosopher, which is expressed in the action A_1. $\oplus$ denotes the modulo operation in GAMMA. R_2 represents the possibility of the release of the forks for a philosopher by A_2. As can be seen, GAMMA defines expressions on multisets and elements for reaction conditions and actions. Here, we refer to the literature for details. In contrast to the LINDA-solution of the problem in 2.1(a), GAMMA ensures fairness in selecting reactions, thus the program is livelock-free.

We can compare GAMMA to ALICE. Where Γ is considered the control-abstraction, ALICE abstracts from any order on the unblocking of agents by making the in-choices non-deterministic. Moreover, Γ also abstracts from order in which the reaction takes place after retrieving a set of molecules, opposed to ALICE's model of sequential processes. The reaction condition differs from the in-mechanism in that it tries to find a set of molecules that fulfill constraints in contrast to the retrieval of a single tuple that is related to a template by the matching rule based on identity of values.

The chemical metaphor of GAMMA has motivated the development of the Chemical Abstract Machine CHAM, which introduces a notion of *subsolutions* in which reactions take place without interfering with other subsolutions. [Berry and Boudol, 90] gives a formal definition of the CHAM-operations and gives CHAM-specification of CCS and a concurrent Λ-calculus, called γ-calculus. The paper emphasizes that the CHAM (and

GAMMA) handles true concurrency as the basic primitive notion in contrast to the named calculi.

Another metaphor is introduced by *Interaction Abstract Machines* ([Andreoli and Ciancarini et al, 92a]), IAM, which try to characterize computations in open systems – as opposed to the focus on parallel computations as with the CHAM.

Here, an agent possesses a state consisting of a multiset of resources. It can evolve when some set of resources is available by replacing it with other resources, as defined in methods of the form $A_1 ⅋ \cdots ⅋ A_n \circ\!\!- B_1 ⅋ \cdots ⅋ B_m$, where the A-resources are replaced by the B-resources. When several non-intersecting subsets of resources are available which are required by multiple methods, they can be replaced concurrently.

The termination of an agent is modeled by rules that contain no resources in the replacement, written $A_1 ⅋ \cdots ⅋ A_n \circ\!\!- \top$. The creation of agents is captured by a notion of cloning. Given a method $p ⅋ a \circ\!\!- (q ⅋ r) \& s$, then on the availability of p and a, two replacement-activities are generated: One in which p and a are replaced by q and r, and one in which they are replaced by s. Thus, there is an intra-agent concurrency by multiple parallel applied methods and inter-agent concurrency by multiple independent replacements in cloned agents.

Communication amongst agents is modeled as the addition of resources to the states of agents. This is done by marking some required resources with the broadcast-prefix $\hat{}$. When such a rule $A_1 ⅋ \cdots ⅋ A_n ⅋ \hat{} C \circ\!\!- B_1 ⅋ \cdots ⅋ B_m$ is executed, then C is assumed to be in the state of the agent and, moreover, is added to the states of all other agents. Broadcasts can be used to model multicasts, if the resources broadcasted are marked by some unique identifier which is required in the methods of a subset of agents only. Then, the resource still is broadcasted but never consumed in some agents. Logically, the agents that know about this identifier are the only receivers of the multicast. IAMs provide a model for computations in the language Linear Objects ([Andreoli and Ciancarini et al, 92b], [Andreoli and Pareschi, 91]).

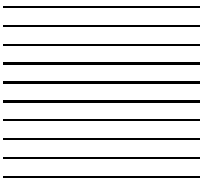
In this chapter we discussed the process synchronization aspect of coordination. We designed the coordination language ALICE which has its own model of execution of processes. We showed, how constructs from a simple imperative language can be implemented with ALICE. Finally, we showed that ALICE can interpret Turing-machines, which documents the orthogonality of computation and coordination.

In the next chapter we present the main contribution of this thesis – the coordination of services in open distributed system with the coordination language LAURA.

> In the case of ALICE we are dealing with a very curious, complicated kind of nonsense, [...] and we need to know a great many things that are not part of the text if we wish to capture its full wit and flavor.
>
> *Martin Gardner's introduction to [Carroll, 60], p. 7*

Chapter 4

Coordination of services in open distributed systems: Laura



In the preceding two chapters we reviewed Linda as a coordination language focussing on the exchange of data in parallel systems and introduced the language Alice which pays special attention to the coordination of processes. In this chapter, we present informally the main contribution of this thesis, the coordination language Laura which is designed for the coordination of services in open distributed systems.

This chapter is organized as follows: First, we give a design rationale by analyzing the characteristics of the coordination problem in open distributed systems and shortly outline Laura's solution. We then describe how services are identified in Laura by descriptions of service-interfaces in section 4.2. The use of names in interfaces in the light of open distributed systems is discussed in section 4.3. Finally, we give a description of Laura's operations in section 4.4. While this chapter gives an informal description, 5 will prescribe Laura formally by a type system and behavioral semantics.

4.1 Design motivation

Open distributed systems provide an infrastructure in which participants use and offer services from and to other participants. They do so at a very large scale – potentially world-wide – and with very few restrictions. The intention is to glue together resources that are already available for some participants but not accessible for all.

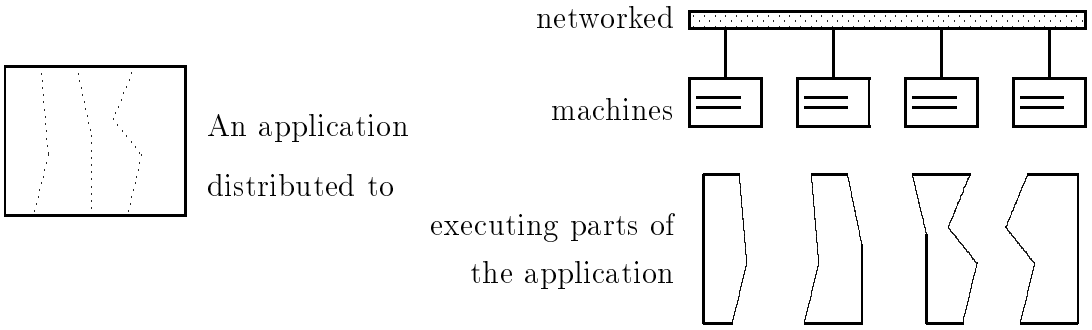


Figure 4.1: Distributing an application

Whereas for a distributed system, a single application is distributed to several networked machines as in figure 4.1, an open distributed system is composed from already existing hardware and software components as in figure 4.2.

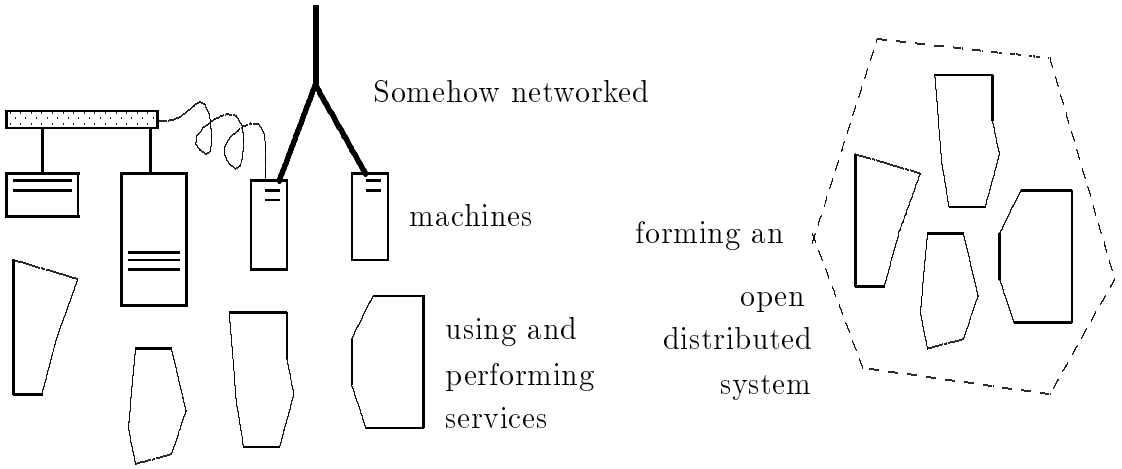


Figure 4.2: Forming an open distributed system

A solution to the coordination problem in open systems has to provide the "glue" that holds together the components. It has to deal with several characteristics of the hardware and software components, such as:

- **Heterogeneity of machine-, network- and operating-system architectures** The machines on which the software components of an open system run, are heterogeneous. That is, they comprise different machine architectures and are from different vendors. They can include personal computers, workstations or mainframes. An open system has to deal with differences amongst them such as byte-orders or value representations.

  The range of operating systems running on these machines can be even broader than that of the hardware architectures. For example, an open system has to

integrate operating systems that support multiple processes, lightweight threads or single processes only. Also, very few assumptions on the outform of file systems can be made. Finally, the networks connecting the systems are heterogeneous, that is they can include different topologies, different protocols and different data-representations on the network.

Coordinating open systems means to deal with these heterogeneities by making them transparent to the user and abstracting from their concrete outforms when designing a coordination system.

- **Heterogeneity of programming languages used for software components**
  The software used in the open system can be programmed in different programming languages. One cannot assume that one language is available across all hardware platforms involved or introduce a potentially world-wide restriction to use one language only.

  The purpose of services coordinated in an open system is not predictable. It may well be that some problems call for the use of a specialized language, or one that is considered more suited than others. Examples could be the use of a parallel Fortran for image-processing, Mumps for database oriented functionalities or C for text-processing functionalities.

  Moreover, with the intention of setting up an open system by integrating pre-existing software-components, a conception that allows the use of multiple programming languages becomes indispensable.

  Coordinating open systems means to abstract from the programming languages used for the software components and to introduce some linguistic means that focus on communication, synchronization, and services. It has to be abstract enough to cope with different models of computation materialized in languages.

- **Potential high dynamics by unrestricted joining and leaving components**
  An open distributed system has no time of beginning or end. It is formed by the components that joined it. In an open system there should be no restriction for components on when they join or when they leave. Examples can be the interactive start of some user interface component, or the replacement of one hard- or software component by another, newer one. No component should be forced to wait for some condition and no component should be hindered from leaving the system by some condition.

  If we understand a failure of an agent as an "unintended leave", this characteristic also covers the requirement that errors of soft- or hardware-component should not inflict other agents in an open system. The recovery of a component can be understood as a "forced join".

  Coordinating open systems means to avoid restrictions and to provide mechanisms that can deal with these dynamics. For example, no assumption on the availability of some component can be made. Even if it is in the system at some time, there is no guarantee that it will not leave quickly thereafter.

As an example, let us look at a hypothetical open system. Say, travel agencies and carriers want to set up a system that enables customers to book and purchase travel tickets from their personal computers at home. Payments made by credit cards have to be authorized by banks. Such a system is appealing as nearly all hard- and software components involved already exist, as bookings probably can be made faster and easier and as some savings and gains are expected by the involved companies[1].

What are the components involved? We can assume, that some personal computers are installed at the homes of potential customers and that they have access to some data lines, say via an ISDN low-band connection provided by a telecommunication carrier. Travel agencies are connected to reservation systems. Such systems use dialup- or leased lines to some central computer system. Their functionality includes queries and the commitment of reservations.

Offers and reservations are transmitted to the carriers using a reservation system and thus connected to the proprietary reservation systems of the carriers. Finally, authorization of credit cards is possible automatically from automata connected to some bank or via an interactive query. Charging credit cards results changes in the customers accounts which take place in some banking system.

Nearly all components for the planned open system are already existing. There is no need to install new terminals such as those in agencies in the homes of potential customers. What is needed is just the "glue" which enables components to cooperate and ensures their sound coordination.

However, the example also shows the characteristics listed above that make up the specifics of the coordination problem in open systems. The personal computer of a customer has to work together with the system of the travel agency. That one, in advance, has to work with a large transaction system in a bank. The machines involved will be of different architectures and will run different operating systems. The network structures range from public telecommunication lines over in-house LANs, as for an agency, or proprietary wide-area-networks, as for banks.

All existing software components can be implemented in very different programming languages. There could be some SQL-driven database of travel-offers and some accounting program written in Cobol. A user interface for the customer could be written in C.

Finally, dynamics arise from customers joining the system at some time to make a booking. Agencies can be established or go out of business, as can carriers.

In this thesis, we describe a design for the "glue" that enables us to coordinate an open system with the characteristics described. It is called LAURA[2] and can be summarized for a first impression as follows.

We understand the software components in an open system as *agents* that use and offer *services* according to their functions. LAURA introduces the abstraction of a *service-space* shared by all agents which is a collection of *forms*. A form can contain a description of a service-offer, a service-request with arguments or a service-result with results.

[1] Which, however, will not lower prices for traveling!

[2] The name is taken from the TV-series "Twin Peaks" by David Lynch, in which the character Laura Palmer is victim of a murder.

The operations of the coordination language LAURA offer linguistic means to put and retrieve offer-, request- and result-forms to and from the service-space. When they are executed, a mechanism similar to LINDA's matching brings together offer-forms and request-forms and delivers the parameters to the service offerer. Result-forms are again matched with requests, so that results can be delivered to the requestor. Figure 4.3 illustrates this abstraction for a set of components involved in the traveling example.

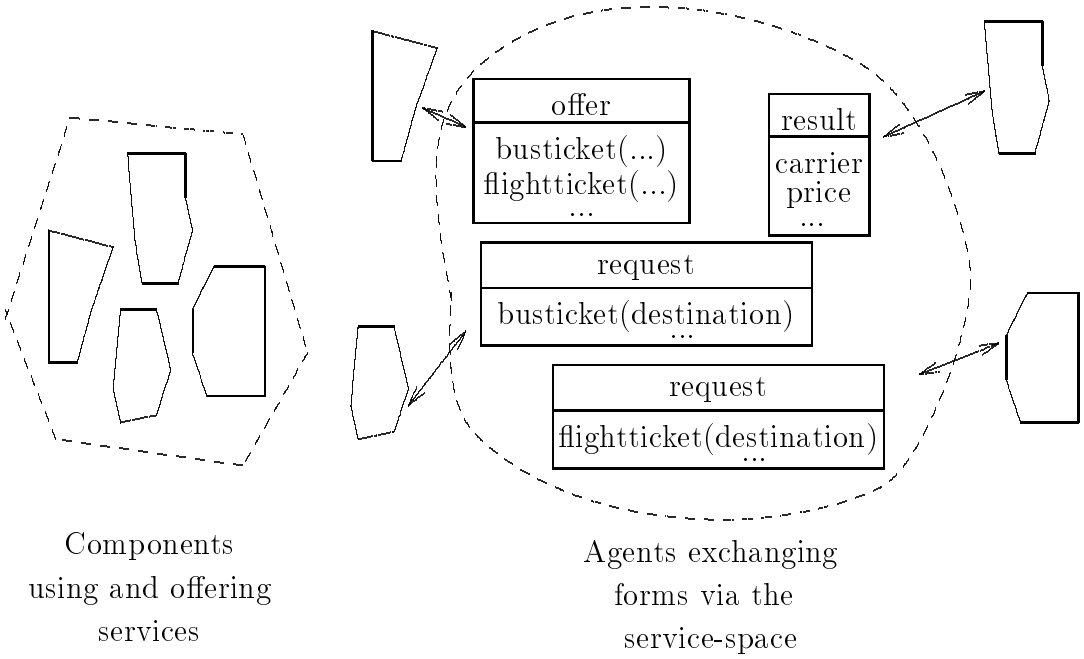


Figure 4.3: A service-space for the traveling example

A main characteristic of our approach shows in the illustration: There are no visible connections amongst agents that offer and request services. An agent knows only about the service-space where it exchanges forms. The service provider and the requestor remain anonymous to each other. We claim that such an *uncoupled* coordination style is a well suited paradigm for open systems because of the following reasons.

In a conventional distributed system, one can assume that – due to its static nature – an agent that performs some service can repeat this at some time in the future. Therefore, it can be efficient to establish a connection between agents for the passing of multiple requests along that connection. If more than one agent offers the same service, it also may be convenient to choose a particular one and to memorize its identifier for further requests.

Both are impossible in open systems, since there is no guarantee that a known agent will be present at some later time. Establishing a connection hinders that agent from leaving the system, which is an unwanted restriction. Memorizing a communication address for later use has the potential of leading to an error, because the agent could have left already.

Moreover, uncoupledness is part of the nature of open systems. There can be multiple offers of similar services by different agents. The decision on which particular service to use should be taken very late on the basis of information that is available at runtime only. This information can change rapidly as the example of communication costs due to communication load shows. With connections, the decision on what agent to communicate with is based on information about the past – the time of the establishment of the connection. This ignores new information such as the availability of a service-offerer which is cheap to communicate with.

The service-space hides the issue of connections from the agents, preventing them from having to cope with joining and leaving agents or with communication addresses and the details of communication. Also, the concrete selection of service providing agents are made by some mechanisms "behind the stages" to which up-to-date information about other agents is available.

LAURA therefore puts emphasis in stating *what service* is requested, not on *which agent* is requested to perform it. A crucial point therefore is the identification of services. The next section explains the approach taken in LAURA.

4.2 Identification of services

The "glue" LAURA uses to coordinate services offered and used by agents in an open distributed system is the exchange of forms via the service-space. As put, forms identify the service requested or offered and necessary information. The question is how to identify a service.

In LAURA, a service is described as an interface consisting of a set of operation signatures. The signatures describe the types of the operations in terms of their names and their argument- and result-types. It is therefore a record of function-types. A form contains a description of this interface-type for service-identification. Putting a service-request form into the service-space starts the search for a service-offer form so that the interface-type of the offer is in a matching relation to that of the request. We do not introduce names for service-types, which is different to what is done in other approaches, such as *direct naming* and *managed types*, as we call them.

In the first, a set of names is defined for services. An example are UNIX system-services that are "named" numerically by some port-number under which services are provided. There is a global convention that, for example, the numeric name 25 identifies the mail-service.

Such an identification scheme is static, as names have to be completely known in advance. ActorSpaces ([Agha and Callsen, 93]) also use a direct naming system, but allow regular expressions to be used for the identification of services. For some services named "`mail`", "`mail-fast`" and "`simple-mail`", the expression "`*mail*`" identifies any one out of those three. However, the name-portion "`mail`" still has to be known in advance. Such a scheme cannot be well-suited for open systems as they are dynamic in nature and as one cannot assume identical naming schemes at a world-wide scale. Our approach is different in that we do not use names for services, but identify them by their interface type solely.

A more dynamic scheme is defined for the ISO standard on open distributed processing ODP ([ISO/IEC JTC1/SC21/WG7, 93b], [ISO/IEC JTC1/SC21/WG7, 94a], [ISO/IEC JTC1/SC21/WG7, 94b], [ISO/IEC JTC1/SC21/WG7, 91][3]). Here, a repository of type definitions is defined which is used to store interface types of services and relations amongst them. A subtype-relation amongst interface types can be explicitly declared or derived from subtyping rules.

Offering or using a service is done via a trading function ([ISO/IEC JTC1/SC21, 93]) that stores offers and their types. It uses the type repository to determine relations amongst offered and requested types and provides a requestor with the identifier of an object that offers an appropriate service. Our approach differs from the ODP scheme in that we do not introduce a repository of types and – as we will see – in that there is no connection between offerers and requestors visible to the agents.

In LAURA, no names are used at all for the identification of services or for the types of data involved in an operation. Instead, a service offered or requested is described by an interface signature consisting of a set of operations signatures. The operation signatures consist of a name and the types of arguments and parameters.

Similar to most approaches to interoperability we use an *interface description language* to notate the interface of a service. This is necessary to facilitate the usage of multiple programming languages. In section 4.5 we give pointers to other approaches to interoperability and to interface description languages involved therein.

In LAURA interfaces are notated in the service type language STL according to the syntax in figure 4.4[4]. To illustrate an interface in STL, we express the type of a service offered or used by a traveling agency. It consists of three operations, `getflightticket`, `getbusticket`, and `gettrainticket` which take as arguments some identification of a credit-card, a travel date and a destination. All operations confirm the purchase and result in a price. `getbusticket` also results in the name of a bus-company. The interface of this service is expressed as:

```
(getflightticket: ccnumber * date * dest -> ack * price;
 getbusticket   : ccnumber * date * dest -> ack * price * line;
 gettrainticket : ccnumber * date * dest -> ack * price)
where
ccnumber = string;
date = <day,month,year>;
day = number;
month = number;
year = number;
dest = string;
ack = boolean;
```

[3]ODP is a standard which is in the balloting phase. The references we give in this thesis refer to those documents that were available at the time of writing. There may be changes in the further stages of standardization that could lead to deviations from our statements. ODP parts 2 and 3 are expected to become international standards in 1995, parts 1 and 4 are expected to reach this status in 1996 ([Raymond, 94]). ODP will become the ISO/IEC standard 10746 and ITU-T recommendations X.901 to X.904.

[4]In the abstract syntax we use a **boldface**-font for terminals.

Service-Type-Declaration::= (Signature-Declaration) **where** Type-Declarations .

Signature-Declaration::= Operation-Signature { ; Operation-Signature }

Operation-Signature::= Operation-Name : Type-Names –> Type-Names

Type-Names::= [Type-Name] { * Type-name }

Type-Declarations::= Type-Declaration { ; Type-Declarations }

Type-Declaration::= Type-name = Type-Definition

Type-Definition::= Predefined-Type | Type-Name |
Type-Definition { * Type-Definition } |
⟨ Type-Definition { , Type-Definition } ⟩ |
[Type-Definition { , Type-Definition } **]**

Predefined-Type::= **string** | **character** | **number** | **boolean**

Figure 4.4: Abstract syntax of service type definitions language STL

```
line = string;
price = <number,number>.
```

In section 5.1 we formally define a type system which is used in the definition of the semantics of such interface definitions in section 5.1.3. This type system includes rules for subtyping and this subtyping is the key for LAURA's identification of services: Given the interface descriptions in forms, a service offer matches a service request, if the type of the interface offered is a subtype of the one requested.

Subtyping in LAURA is defined so that a type A is a subtype of B if all values of type A can be substituted when a value of type B is requested. The "values" we type are services. The subtyping enables us to use a service of type A if a service of type B is requested.

For the traveling example, the typing makes it possible to have an agency that offers bus-, train- and flight-tickets perform the purchase of a train-ticket when an agency is requested that offers bus- and train-tickets. It also rules out agencies offering bus- and flight-tickets to be selected for the purchase. When following interface description is contained in a service-form, the form matches a serve-form with the interface above, as their types are in a subtype-relation according to our type system:

```
(getflightticket: ccnumber * date * dest -> ack * price;
 gettrainticket : ccnumber * date * dest -> price)
where
ccnumber = string;
date = <day,month,year>;
day = number;
```

```
month = number;
year = number;
dest = string;
ack = boolean;
price = <number,number>.
```

But before we formalize the type system and STL's semantics in chapter 5, we have to discuss the issue of naming in open distributed systems more detailed, as we will have to cope with names, as the examples show at least for the operation-names. This is the topic of the next section.

4.3 On naming in open distributed systems

Above we stated informally that a service interface type consists of a number of operation signatures formed by an operation name and lists of types for arguments and results. A relation on operation signatures – such as a subtype relation – will take the operation names into account. An interface signature then can be understood as a record consisting of tagged fields, where the tag corresponds to the operation name and the value to the type of the function.

Common equivalence relations for tagged records require all tags and their associated values to be equal. A subtyping-relation as in [Cardelli, 88], [Amadio and Cardelli, 91], requires that all fields of the supertype have to have corresponding fields in the subtype with identical tags and values being in a subtype relation to those of the supertype. Examing such a rule closer shows that type-checking involves some form of name checking, too. In fact, for the given rule this name-checking means testing for syntactic equivalence.

Say, flight-carriers register some information about a traveler at check-in[5]. At Heathrow Airport, London, this information is kept in a record A=⟨baggagetag: string, withchild: boolean, smoker: boolean⟩. For a transatlantic flight this information is passed to the destination airport, which could be JFK, New York. Information about passengers there is stored in records of the type B=⟨luggagetag: string, withchild: boolean⟩.

We have to test, if A conforms to B when the information is to be transferred so that the record can be stored safely in the database at JFK. A short glance at the record makes it clear that it should conform as we can forget the entry about smoking and as we know that "baggagetag" and "luggagetag" mean the same thing. Applying formal subtyping rules, however, does not identify A as being a subtype of B as the names are syntactically distinct. There is a discrepancy between what we informally state and what is yielded by the formal test.

[5]In fact, for reasons of safety, more and more airports install systems (baggage reconciliation systems) that assign a number to each baggage item which is printed in machine-readable form on a tag. Prior to loading the baggage to the aircraft, this information is read by some mobile device and used to verify that the owning passenger has passed the check-in to make sure that no baggage is transported without its owner. A spokesman of Frankfurt Airport stated that the major problems for installing this system were posed by the sheer number of baggage and passengers handled and by the complexity of the baggage-transportation system ([Jopp, 94]).

We can represent a naming system by relations between names and semantic objects, where objects are represented by names. We call this relation $\mathcal{R}$ and write $n\mathcal{R}o$ when the name n represents the object o.

Names can be understood as an encoding of the intended meaning in identifying objects and the objects themself as the extension of this identification. When we consider a naming system in which all semantic objects are identified by one single name and a name always identifies exactly one semantic object, we can justify a formal rule that requires a syntactical equivalence on names. Such a system satisfies an axiom of extensionality for $n = m$ as $n\mathcal{R}o_1 \wedge m\mathcal{R}o_2 \Rightarrow o_1 = o_2$ holds.

The example above illustrates that this assumption can be too restrictive as it disallows "baggagetag" and "luggagetag" to identify the same semantic object. For an open system in which global and consistent knowledge is absent, the same names can well be used for different semantic objects and different names can refer to the same object.

Let $\mathcal{M}$ be a relation on names that is used to determine if two names refer to the same semantic object. Four possibilities with respect to $\mathcal{R}$ arise for objects o_1 and o_2 and names n and m as follows:

- Accidental mismatch $m\mathcal{R}o_1 \wedge n\mathcal{R}o_1 \wedge \neg(m\mathcal{M}n)$
- Intended match $m\mathcal{R}o_1 \wedge n\mathcal{R}o_1 \wedge m\mathcal{M}n$
- Intended mismatch $m\mathcal{R}o_1 \wedge n\mathcal{R}o_2 \wedge \neg(m\mathcal{M}n)$
- Accidental match $m\mathcal{R}o_1 \wedge n\mathcal{R}o_2 \wedge m\mathcal{M}n$

We can define two properties: $\mathcal{M}$ is said to be *sound* if $m\mathcal{M}n \Rightarrow m\mathcal{R}o \wedge n\mathcal{R}o$ and to be *complete* if $m\mathcal{R}o \wedge n\mathcal{R}o \Rightarrow m\mathcal{M}n$. Soundness states that no accidental match will occur and completeness that no accidental mismatch occurs.

$\mathcal{R}$ is an abstract relation, today we have no means to derive $\mathcal{M}$ from it by some algorithm from a given set of names and objects. However, we can reason about the appropriateness of relations on names. We identify three outforms of $\mathcal{M}$ that seem reasonable:

1. $n\mathcal{M}m \Leftrightarrow n = m$. This is the syntactic relation on names which we referred to above.

2. $n\mathcal{M}m \Leftrightarrow s(n) = s(m)$. Here, s is a function on names that can provide some structural information. For a record, s could be defined as the position of a name in a record.

3. $n\mathcal{M}m$ = TRUE. Here, names are ignored by defining all names as pairwise matching.

Note that all three forms are not sound, as accidental matches can occur. The last relation is complete, as no accidental mismatches can occur. The difference between form one and three represents the semantics of names, the difference between two and three the semantics of ordering.

For LAURA, we chose the first form for interface-types where the names of operations offered have to be syntactical equivalent. We will use the last outform for any names that occur in the types used as arguments or results for operations. This decision is a

compromise between completeness – wanting to accept the flight-information example – and soundness – avoiding to increase the quantitative amount of accidental matches. The type system we define in chapter 5 includes rules to express this choice in the formal definition of the semantics of STL.

After having now explained how services are identified in LAURA by an interface of operations and without a global naming system, we now detail out LAURA's operations that are used by agents to put and retrieve forms into and from the service-space and how they are executed.

4.4 LAURA's operations

In 4.1 we briefly described LAURA's operations that coordinate agents in an open system by the exchange of forms via the service-space. In the previous sections, we defined how services are identified on these forms. Now, we define LAURA's operations in detail.

In the examples, we assume that the interfaces with which we illustrated STL above are abbreviated by some name which is locally known to agents and which should not be misinterpreted as a service-name or global identifiers:

small-agency=

```
(getflightticket:
  ccnumber * date * dest ->
  ack * price;
 gettrainticket :
  ccnumber * date * dest -> price)
where
ccnumber = string;
date = <day,month,year>;
day = number;
month = number;
year = number;
dest = string;
ack = boolean;
price = <number,number>.
```

large-agency=

```
(getflightticket:
  ccnumber * date * dest ->
  ack * price;
 getbusticket :
  ccnumber * date * dest ->
  ack * price * line;
 gettrainticket :
  ccnumber * date * dest ->
  ack * price)
where
ccnumber = string;
date = <day,month,year>;
day = number;
month = number;
year = number;
dest = string;
ack = boolean;
line = string;
price = <number,number>.
```

A service is the result of an interaction between a service-provider and a service-user. In LAURA, two operations coordinate this interaction for the service-provider, `serve` and `result`.

An agent that is willing to offer a service to other agents puts a serve-form into the service-space. It does so by executing `serve`, which takes as parameters the type of the service offered and a list of binding rules that define to which program variables arguments for the service should be bound. For the example *large-agency*, the operation would be formulated as

```
SERVE large-agency operation
(getflightticket: cc * <day,month,year> * dest -> ack * <dollar,cent>;
 getbusticket   : cc * <thedate.day,thedate.month,thedate.year> * dest ->
                  ack * <dollar,cent> * line;
 gettrainticket : cc * <day,month,year> * dest -> ack * <dollar,cent>).
SERVE
```

This states that a service with the interface *large-agency* is offered and that a code for the selected operation should be bound to the program variable `operation`. In the case of the operation `gettrainticket`, the arguments provided by the service-user should be bound to the program variables `cc`, `day`, `month`, `year` and `dest`. Note that in contrast to the names used only for convenience in the definition of a service-interface, the names used in the binding lists are those of variables that have to be declared properly in the program text of the agent. The names used in the result-parts of the operations are ignored.

When a `serve` is executed, a serve-form is built from the arguments. Then, the service-space is searched for a service-request form whose service-type matches the offered service by being a supertype. The code of the requested operation and the provided arguments are copied to the serve-form and finally bound to program variables according to the binding list. The `serve`-operation blocks as long as no matching request-form is found.

After performing the service requested, the service-provider uses `result` to deliver a result-form to the service-space. This operation looks similar to `serve`:

```
RESULT large-agency operation
(getflightticket: cc * <day,month,year> * dest -> ack * <dollar,cent>;
 getbusticket   : cc * <thedate.day,thedate.month,thedate.year> * dest ->
                  ack * <dollar,cent> * line;
 gettrainticket : cc * <day,month,year> * dest -> ack * <dollar,cent>).
RESULT
```

Here, the names used in the argument parts of the binding lists are ignored. A result-form is built which consists of the service-interface and – depending on `operation` – a list of result values according to the binding list. For the case of a `gettrainticket`, they are taken from the program variables `ack`, `dollar` and `cent`. The agent is responsible to store the results of the service properly in those variables. The operation is performed immediately and the form is put into the service-space.

An agent that offers services usually operates in a loop consisting of the sequence `serve`–*perform the service*–`result`. However, LAURA makes no assumptions on this behavior nor enforces it. This is due to the fact that no assumptions on the programming

language used for the agent and its execution model can be made. It can well be the case that multiple services are performed concurrently or that the order of service provision and result-delivery does not match the order of service-requests.

An agent that wants to use a service has to execute LAURA's third and last operation, `service`. Its arguments are the service-type requested, the operation requested, arguments for the operation and a binding-list. An example is

```
SERVICE small-agency
(getflightticket : cc * <thedate.day,thedate.month,thedate.year> * dest ->
                   ack * <dollar,cent>;).
SERVICE
```

Here, a service with an interface *small-agency* is requested. The operation to be performed is `getflightticket`. The binding lists from both the argument- and result-part are used to access the arguments stored in the program variables `cc`, `thedate.day`, `thedate.month`, `thedate.year` and `dest`. The results of the service should be bound to `ack`, `dollar` and `cent`.

Up to now, we talked about a service-request form for the sake of simplicity. In fact, when executing `service`, two forms are involved: a service-put form and a service-get form. The first is constructed from the service-interface and the arguments and then inserted to the service-space. If another agent performs a `serve`-operation and the service-put- and serve-forms match, the arguments are copied as described above and the service-provider starts processing the requested operation.

The service-get form is constructed from the service interface and the binding list for the results. Then, a matching result-form is sought in the service-space and – when available – the results are copied and bound to the program variables.

When the request-form is entered to the service-space, it is matched with some serve-form, thus starting the execution of the requested service by some agent. When the result-form is retrieved, the results are bound to the local environment according to the binding list.

The interaction of agents coordinating services with LAURA consists either of putting a request for a service to the service-space, finding a matching offer form and copying of arguments or of trying to get the results of a service, by finding a matching result-form and copying of the results. This interaction is uncoupled, as service-provider and -user remain completely anonymous to each other. However, there remains the problem, that a `result` provides the results for a specific `service`.

Given that we have only two agents working on the service-space, the interaction as described will always succeed. When there are more service-users and -providers, the case can arise that two identical services are requested and result-forms for them are emitted by providers – or a single provider that processes services concurrently – to the service-space. In this case we want that the results of a service are given to the agent that requested it – which cannot be achieved if the interaction is implemented completely uncoupled and based on the matching of service-types only.

"Behind the stages" of LAURA there has to be some mechanism that turns the logical uncoupling into a concrete coupling for the period of time between the choice of

some service-provider, the invocation of operations of this agent and the delivery of the service-effects – i.e. the results – to the service-user.

It does so by *form-transformations* that result in *unique forms* by the addition of some unique identifiers. When `service` is performed, LAURA generates this unique identifier and extends the form by it. `serve` stores this identifier within a LAURA-library – that has to be used by any agent – and extends the `result`-form with it. In this case there is only one result-form with that identifier and it can be retrieved by `service`. The resulting logical connection between provider and requestor of a service is bound to the forms and does not require unique identifiers for agents. The logical connection does not imply a physical connection such as a communication channel but is manifested by the unique identifier in the form that then is used for the matching of unique forms.

Figure 4.5 on page 52 shows a service interaction and the forms involved. The providing agent executes a `serve` with the offered interface `I`, a placeholder for the operation code `?o` and binding-rules for arguments `?a`. The library transforms this serve-form into one with a placeholder for a unique-identifier `u` prepended.

The service-user executes a `service` for a service with interface `J`, operation `o` with arguments `a` and binding-rules for the results `?r`. The library generates unique put- and get-forms from these which include the unique identifier `u`. Then a match occurs when `I` is a subtype of `J` in which case the arguments and `o` are copied. The put-form is removed from the service-space and the serve-form delivered to the provider with arguments filled in with `a`. The library strips off `u`, stores it and binds the values from `o` and `a` according to the binding rules `?o` and `?a`.

The provider processes the service and performs a `result` with results `r`. The library prepends the stored `u` and inserts the unique result-form to the service-space. Here a second match occurs with the unique get-form and the results `r` are copied. The result-form is destroyed and the put-form delivered to the service-user. Here the library strips off the unique identifier and binds `r` according to the binding-rules `?r`.

The following list highlights the characteristics of LAURA and how they meet the requirements of service-coordination in open distributed systems:

- **Separated focus on coordination** LAURA does focus on the coordination of services and introduces a complete language with respect to this task. Thereby, no assumptions are made on programming languages the implement the processing of services or their execution models. This is necessary to allow the use of multiple languages in an open system.

- **Uncoupled coordination** LAURA requires no form of coupling amongst service-user and -provider. The logical connection with unique identifiers is hidden and induces no physical connections. This is necessary to cope with the dynamics of joining and leaving agents in open systems.

- **Service identification by typed interfaces and subtyping** In LAURA services are identified by the type of their interfaces solely and selected based on a subtyping relation. This is necessary to avoid a global naming mechanisms and to make use of multiple offers for similar services.

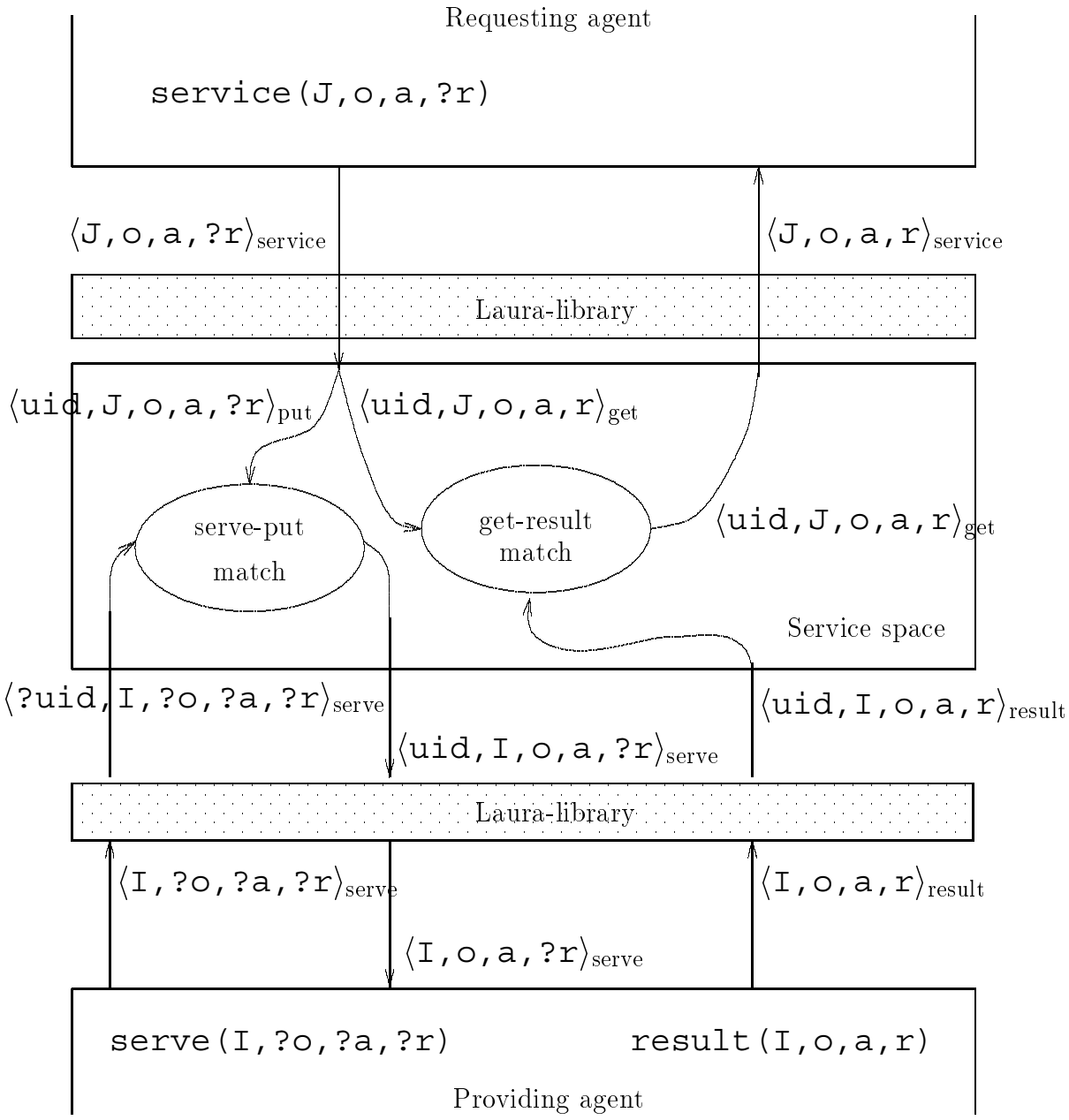


Figure 4.5: Forms involved when coordinating a service

With these we take into account the characteristics we listed in the first section of this chapter.

4.5 Bibliographic remarks

A variety of projects have attacked the problem of coordination in open and heterogeneous environments. We name two of them.

At the University of Maryland, a project developed a system called Polylith Software Bus which focusses on the use of modules written in different languages by defining interfaces for their functionality and connecting them via a software bus that transports invocations in a distributed, heterogeneous environment ([Purtilo and Jalote, 89], [Callahan and Purtilo, 90], [Purtilo, 90]. The work investigated in how interfaces can be safely adapted when the functionality changes ([Purtilo and Atlee, 91]), on how the types of interfaces can be checked ([Myers and Purtilo, 92]) and how mapping to programming languages can be implemented ([Shannon and Snodgrass, 89]). In contrast to Laura, Polylith is designed for a distributed environment and does not pay special attention to the requirements of open systems.

Another early approach to install an open distributed system was undertaken in a joint project of the Technical University Berlin together with the Deutsche Herzzentrum Berlin, called Heterogeneous Document Management System HDMS ([Hansen and Kutsche et al, 91], [Hansen and Kutsche, 94]). It focusses on the integration of patient files that can contain numerical medical examination data, textual documents such as reports, graphical information such as X-ray images or video films from heart catheter examinations.

In HDMS all documents are represented as objects, which encapsule data, processes that manipulate the data and an interface, on which methods are invoked which are executed as processes. Underlying the implementation is the "Objekt-Maschine" which provides the glue to integrate various medical and computational hardware connected by different networks and driven by specialized software components – such as medical image processing – in a single medical workplace.

Components of HDMS include, amongst others, an interface description and object specification language called DIDO ([Kutsche, 94]), an object-oriented role model which is used to give a structured description of people using HDMS such as doctors or nurses and to infer access-right to objects from it ([Gayda, 92]), and a transaction system ([Wittkugel, 94]). In contrast to LAURA, service coordination is performed by a remote object method call mechanism which involves coupling amongst objects.

The problem-space of naming has been analyzed in a taxonomy of issues in name-systems in [Yeo and Ananda et al, 93]. The notions of soundness and completeness can be found in [Bowman and Debray et al, 93], where they are used as tools to establish hierarchies of name-resolution functions. Distinguishing name-objects and types as intention and extension to reason about names is done on a formal basis in [Marzetta, 92].

In this chapter we gave an informal description of the coordination language LAURA which introduces a service-space which is used for the coordination of services in open distributed systems. We introduced three operations that put and withdraw forms into and from the service-space. They are `serve` which puts a service-offer form into the service-space which is returned filled with an operation code and arguments. `result` puts a service-result form into the space in which the results are filled out. `service`,

finally, is used by a service-user where the requested operation and arguments are filled out and which is returned with the results of the service.

Form-transformations implement a logical connection hidden from the agents which remain anonymous to each other. Services are identified by the types of the service-interfaces and the matching of forms depends on a subtype-relation on these interfaces.

An informal description requires a formal prescription to identify implementations that are considered correct. This is the topic of the next chapter in which we formally define a type system to interpret service-interfaces, introduce a model of coordination by manipulation of multisets which we use to give a specification of the behavior of agents using LAURA's operations.

Chapter 5

Prescribing Laura formally

Service coordination in open distributed systems with Laura has been described informally in the previous chapter. When implementing such a system, a formal definition is needed that defines constraints that identify correct implementations. This chapter gives these prescriptive formal definitions before an experimental implementation is discussed in the next chapter.

Formal methods serve to bridge the gap between an informal description of a system and its implementation. The formalizations in this chapter focus on the main components of Laura, but do not define a complete formal definition of the language. Our intention is to prescribe correct implementations as abstract as possible and not to gain theoretical insight.

This chapter consists of two main formal definitions. As described, service identification in Laura is based on typed interfaces. In section 5.1 we define a type system which reflects the special considerations on naming of section 4.3. This type system is used to give semantics to STL declarations in section 5.1.3.

The second part is on the behavior of Laura's operations. We define a machine – the Bag-Machine – in section 5.2 which performs coordination with a multiset of some elements. We give semantics to the Bag-Machine operations by a labeled event structure in section 5.4 and to agents utilizing the Bag-Machine in section 5.5. After using the Bag-Machine to specify Linda as an example, we finally use it to define the behavior of Laura's operations in section 5.8.

5.1 A type system with subtyping

LAURA's matching rule uses a type system for service interfaces. Typing of services is a common approach which can also be found in ODP, for example. Using a type system with subtyping is appropriate for open distributed systems as it allows a service-user to identify what sort of service is requested instead of stating what agent is requested to perform a service. Moreover, subtyping is a convenient mechanism to deal with multiple "similar" service-offers, such as specializations of existing services.

The type system we use for LAURA is defined by a set of inference rules for type equivalence- and subtyping-relations. It consists of constants, three record-types with different treatment of names, products, three union-types with different matchings for names and function types. Type terms are taken from a language which is generated by the following grammar, where $a_1, \ldots, a_n$ are names and $\alpha_1, \ldots, \alpha_n$ and β type terms.

$$\alpha ::= t \mid \langle a_1\colon \alpha_1, \ldots, a_n\colon \alpha_n\rangle \mid \langle a_1\colon \alpha_1, \ldots, a_n\colon \alpha_n\rangle_O \mid \langle a_1\colon \alpha_1, \ldots, a_n\colon \alpha_n\rangle_A \mid \alpha_1 \times \ldots \times \alpha_n \mid [a_1\colon \alpha_1, \ldots, a_n\colon \alpha_n] \mid [a_1\colon \alpha_1, \ldots, a_n\colon \alpha_n]_O \mid [a_1\colon \alpha_1, \ldots, a_n\colon \alpha_n]_A \mid \alpha \to \beta.$$

The different outforms of records and unions reflect the different forms of name-treatment as described in section 4.3. $\langle a_1\colon \alpha_1, \ldots, a_n\colon \alpha_n\rangle$ is called an *exact record*, in which the names are used for a syntactic matching, $\langle a_1\colon \alpha_1, \ldots, a_n\colon \alpha_n\rangle_O$ an *ordered record*, where the names are unimportant and only the structural information on the order of fields is used. When working with an *anonymous record* – $\langle a_1\colon \alpha_1, \ldots, a_n\colon \alpha_n\rangle_A$ – both the syntactic and structural properties of the names are neglected. The rules below will make it clearer what exactly this means. We start with the rules for type equivalence.

5.1.1 Rules for type-equivalence

Equivalence of types is defined by the inference rules in figure 5.1 on page 57. They establish an equivalence relation $=$, which is reflexive (EREFL), symmetric (ESYM) and transitive (ETRAN).

For the records-types, the inference rules reflect the different outforms of a matching on names which we outlined in the section 4.3 on naming in open distributed systems. For ERECEX all names of the record fields have to be exactly matching, that is they have to be syntactical equivalent. All types of the fields have to be pairwise identical. ERECORD uses the structural information given by the position of a field as the matching criteria for names for an ordered record. All field names are renamed to their position and these renamed exact records have to be equivalent. Finally, ERECANON discards the name-information completely. Two anonymous records are equivalent if one can be permuted so that the permutation as an ordered record is equivalent to the other. The scheme $(\ldots)$ denotes the set of permutations over a record.

EPROD infers equivalence of products from the equivalence on anonymous records. EUNIEX, EUNIORD and EUNIANON define equivalence for exact, ordered and anonymous unions similar to those for records. EUNIPERM reflects the fact that unions do not carry structural information per se. A union-type stands for the unordered union of fields, so that permutation of fields is allowed when inferring the equivalence of unions. As the last rule, EFUNC defines equivalence of function types as the equivalence of argument and result types.

$$\frac{}{\alpha = \alpha}\ \textsc{ERefl} \qquad \frac{\alpha = \beta}{\beta = \alpha}\ \textsc{ESym} \qquad \frac{\alpha = \beta \quad \beta = \gamma}{\alpha = \gamma}\ \textsc{ETran}$$

$$\frac{\forall i \in \{1, \ldots, n\} : \alpha_i = \beta_i}{\langle a_1 \colon \alpha_1, \ldots, a_n \colon \alpha_n \rangle = \langle a_1 \colon \beta_1, \ldots, a_n \colon \beta_n \rangle}\ \textsc{ERecEx}$$

$$\frac{\langle 1 \colon \alpha_1, \ldots, n \colon \alpha_n \rangle = \langle 1 \colon \beta_1, \ldots, n \colon \beta_n \rangle}{\langle a_1 \colon \alpha_1, \ldots, a_n \colon \alpha_n \rangle_O = \langle b_1 \colon \beta_1, \ldots, b_n \colon \beta_n \rangle_O}\ \textsc{ERecOrd}$$

$$\frac{\langle a_x \colon \alpha_x, \ldots, a_y \colon \alpha_y \rangle_O = \langle b_1 \colon \beta_1, \ldots, b_n \colon \beta_n \rangle_O, \langle a_x \colon \alpha_x, \ldots, a_y \colon \alpha_y \rangle \in \left(\begin{smallmatrix} \langle a_1 : \alpha_1, \ldots, a_m : \alpha_m \rangle_O \\ \langle a_{i_1} : \alpha_{i_1}, \ldots, a_{i_m} : \alpha_{i_m} \rangle_O \end{smallmatrix} \right)}{\langle a_1 \colon \alpha_1, \ldots, a_n \colon \alpha_n \rangle_A = \langle b_1 \colon \beta_1, \ldots, b_n \colon \beta_n \rangle_A}\ \textsc{ERecAnon}$$

$$\frac{\langle 1 \colon \alpha_1, \ldots, n \colon \alpha_n \rangle_A = \langle 1 \colon \beta_1, \ldots, n \colon \beta_n \rangle_A}{\alpha_1 \times \ldots \times \alpha_n = \beta_1 \times \ldots \times \beta_n}\ \textsc{EProd}$$

$$\frac{\forall i \in \{1, \ldots, n\} : \alpha_i = \beta_i}{[a_1 \colon \alpha_1, \ldots, a_n \colon \alpha_n] = [a_1 \colon \beta_1, \ldots, a_n \colon \beta_n]}\ \textsc{EUniEx}$$

$$\frac{[a_x \colon \alpha_x, \ldots, a_y \colon \alpha_y] = [b_1 \colon \beta_1, \ldots, b_n \colon \beta_n] \quad [a_x \colon \alpha_x, \ldots, a_y \colon \alpha_y] \in \left(\begin{smallmatrix} [a_1 : \alpha_1, \ldots, a_m : \alpha_m] \\ [a_{i_1} : \alpha_{i_1}, \ldots, a_{i_m} : \alpha_{i_m}] \end{smallmatrix} \right)}{[a_1 \colon \alpha_1, \ldots, a_n \colon \alpha_n] = [b_1 \colon \beta_1, \ldots, b_n \colon \beta_n]}\ \textsc{EUniPerm}$$

$$\frac{[1 \colon \alpha_1, \ldots, n \colon \alpha_n] = [1 \colon \beta_1, \ldots, n \colon \beta_n]}{[a_1 \colon \alpha_1, \ldots, a_n \colon \alpha_n]_O = [b_1 \colon \beta_1, \ldots, b_n \colon \beta_n]_O}\ \textsc{EUniOrd}$$

$$\frac{[a_x \colon \alpha_x, \ldots, a_y \colon \alpha_y]_O = [b_1 \colon \beta_1, \ldots, b_n \colon \beta_n]_O, [a_x \colon \alpha_x, \ldots, a_y \colon \alpha_y] \in \left(\begin{smallmatrix} [a_1 : \alpha_1, \ldots, a_m : \alpha_m]_O \\ [a_{i_1} : \alpha_{i_1}, \ldots, a_{i_m} : \alpha_{i_m}]_O \end{smallmatrix} \right)}{[a_1 \colon \alpha_1, \ldots, a_n \colon \alpha_n]_A = [b_1 \colon \beta_1, \ldots, b_n \colon \beta_n]_A}\ \textsc{EUniAnon}$$

$$\frac{\alpha_1 = \alpha_2 \quad \beta_1 = \beta_2}{\alpha_1 \to \beta_1 = \alpha_2 \to \beta_2}\ \textsc{EFunc}$$

Figure 5.1: Rules for type-equivalence

5.1.2 Rules for subtyping

The rules for subtyping infer judgements on a subtype-relation by interpreting the rules in figure 5.2 on page 58 relative to an environment Γ which contains assumptions on subtype relations. SAss defines how these judgements are made.

Γ denotes a set $\{t_1 \lessdot s_1, \ldots, t_n \lessdot s_n\}$ of subtyping assumptions on type variables from which judgements on subtype relations are based (SAss). With rule SRefl, SAsym and STrans together with the minimal type $\bot$ – no value is of this type – (SMin) and $\top$ – all values are of this type – (SMax), $\leqslant$ is a partial order.

An exact record is a subtype of another if the types of fields with identical names are in the subtype-relation (SRecEx). The subtype can have more fields, making it possible to substitute a value of the subtype for its supertype by forgetting the additional fields.

$$\frac{t \leqslant s \in \Gamma}{\Gamma \vdash t \leqslant s}\ \textsc{SAss} \qquad \frac{\Gamma \vdash \alpha \leqslant \beta \quad \Gamma \vdash \beta \leqslant \alpha}{\alpha = \beta}\ \textsc{SAsym} \qquad \frac{\Gamma \vdash \alpha \leqslant \beta \quad \Gamma \vdash \beta \leqslant \gamma}{\Gamma \vdash \alpha \leqslant \gamma}\ \textsc{STrans}$$

$$\frac{\alpha = \beta}{\Gamma \vdash \alpha \leqslant \beta}\ \textsc{SRefl} \qquad \frac{}{\Gamma \vdash \bot \leqslant \alpha}\ \textsc{SMin} \qquad \frac{}{\Gamma \vdash \alpha \leqslant \top}\ \textsc{SMax}$$

$$\frac{\forall i \in \{1, \ldots, n\} : \Gamma \vdash \alpha_i \leqslant \beta_i \quad n \leq m}{\Gamma \vdash \langle a_1 \colon \alpha_1, \ldots, a_m \colon \alpha_m \rangle \leqslant \langle a_1 \colon \beta_1, \ldots, a_n \colon \beta_n \rangle}\ \textsc{SRecEx}$$

$$\frac{\Gamma \vdash \langle 1 \colon \alpha_1, \ldots, m \colon \alpha_m \rangle \leqslant \langle 1 \colon \beta_1, \ldots, n \colon \beta_n \rangle \quad n \leq m}{\Gamma \vdash \langle a_1 \colon \alpha_1, \ldots, a_m \colon \alpha_m \rangle_O \leqslant \langle a_1 \colon \beta_1, \ldots, a_n \colon \beta_n \rangle_O}\ \textsc{SRecPos}$$

$$\frac{\begin{array}{l}\Gamma \vdash \langle a_1 \colon \alpha_1, \ldots, a_i \colon \alpha_i \rangle_O <^{\circ} \langle b_1 \colon \beta_1, \ldots, b_j \colon \beta_j \rangle_O \quad j \leq i, j < l < n \\ \Gamma \vdash \langle a_k \colon \alpha_k, \ldots, a_m \colon \alpha_m \rangle_O \leqslant \langle b_l \colon \beta_l, \ldots, b_n \colon \beta_n \rangle_O \quad i < k < m, (n-l) \leq (m-k)\end{array}}{\Gamma \vdash \langle a_1 \colon \alpha_1, \ldots, a_i \colon \alpha_i, \ldots, a_k \colon \alpha_k, \ldots, a_m \colon \alpha_m \rangle_O \leqslant \langle b_1 \colon \beta_1, \ldots, b_j \colon \beta_j, b_k \colon \beta_k, \ldots, b_n \colon \beta_n \rangle_O}\ \textsc{SRecOrd}$$

$$\frac{\Gamma \vdash A \leqslant \langle b_1 \colon \beta_1, \ldots, b_n \colon \beta_n \rangle_O \quad A \in \binom{\langle a_1 : \alpha_1, \ldots, a_m : \alpha_m \rangle_O}{\langle a_{i_1} : \alpha_{i_1}, \ldots, a_{i_m} : \alpha_{i_m} \rangle_O} \quad n \leq m}{\Gamma \vdash \langle a_1 \colon \alpha_1, \ldots, a_m \colon \alpha_m \rangle_A \leqslant \langle b_1 \colon \beta_1, \ldots, b_n \colon \beta_n \rangle_A}\ \textsc{SRecAnon}$$

$$\frac{\Gamma \vdash \langle 1 \colon \alpha_1, \ldots, n \colon \alpha_n \rangle_A \leqslant \langle 1 \colon \beta_1, \ldots, n \colon \beta_n \rangle_A}{\Gamma \vdash \alpha_1 \times \ldots \times \alpha_n \leqslant \beta_1 \times \ldots \times \beta_n}\ \textsc{SProd}$$

$$\frac{\forall i \in \{1, \ldots, n\} : \Gamma \vdash \alpha_i \leqslant \beta_i \quad n \leq m}{\Gamma \vdash [a_1 \colon \alpha_1, \ldots, a_n \colon \alpha_n] \leqslant [a_1 \colon \beta_1, \ldots, a_m \colon \beta_m]}\ \textsc{SUniEx}$$

$$\frac{\Gamma \vdash [1 \colon \alpha_1, \ldots, n \colon \alpha_n] \leqslant [1 \colon \beta_1, \ldots, m \colon \beta_m] \quad n \leq m}{\Gamma \vdash [a_1 \colon \alpha_1, \ldots, a_n \colon \alpha_n]_O \leqslant [a_1 \colon \beta_1, \ldots, a_m \colon \beta_m]_O}\ \textsc{SUniOrd}$$

$$\frac{\Gamma \vdash A \leqslant [b_1 \colon \beta_1, \ldots, b_m \colon \beta_m]_O \quad A \in \binom{[a_1 : \alpha_1, \ldots, a_n : \alpha_n]_O}{[a_{i_1} : \alpha_{i_1}, \ldots, a_{i_n} : \alpha_{i_n}]_O} \quad n \leq m}{\Gamma \vdash [a_1 \colon \alpha_1, \ldots, a_n \colon \alpha_n]_A \leqslant [b_1 \colon \beta_1, \ldots, b_m \colon \beta_m]_A}\ \textsc{SUniAnon}$$

$$\frac{\Gamma \vdash \alpha' \leqslant \alpha \quad \Gamma \vdash \beta \leqslant \beta'}{\Gamma \vdash \alpha \to \beta \leqslant \alpha' \to \beta'}\ \textsc{SFunc}$$

Figure 5.2: Rules for subtyping

Rule SRecPos defines the subtype relation for records ordered by the positions of fields. Here the names are replaced by their position in the record, thus using structural information for the matching on names. In contrast to the exact matching, additional fields can occur only at the end of the subtype.

Thus, the structural information "position" is too strong to what we intend. Therefore, we add rule SRecOrd allowing additional fields at any place of the subtype as long as the order of the fields also existent in the supertype is obeyed. The structural information we use here is "order" instead of "position" only.

Finally, SRecAnon discards names and structural information by defining subtyping on anonymous records by requiring one permutation of the subtype to be in the

ordered subtype relation. As with the equivalence relation, subtyping on products is inferred from the subtype relation on anonymous records.

For unions to be in a subtype relation that ensures substitutability, the subtype may not have more variants than the supertype. SUniEx takes the syntactic name matching for exact unions into account. For ordered unions, SUniOrd discards the names but uses the positions of the variants. SUniAnon allows permutations to be used for anonymous unions.

For function types, we define a contravariant subtyping by rule SFunc. A function type is a subtype of another when its arguments are supertypes to those of its supertype and the results are subtypes to those of the supertype. By this, a subtyped function can safely replace its supertype function by discarding the additional arguments and by forgetting the additional results.

The type system we defined is underlying Laura's service type concept. The semantics of expressions from Laura's service type definition language as defined in figure 4.4 in section 4.2 are given in the next section.

5.1.3 The semantics of Laura's service-type definitions

The semantics of a type expression is defined by interpreting it as a type in the above type system. The interpretation is relative to a finite set Υ of type definitions which associate type variables to types. Υ is required to be wellformed, meaning that all type-variables of the expression have to be in the domain of Υ.

The environment Υ of an interface type contains Laura's ground types **string**, **character**, **number**, and **boolean** together with those type declarations found in the **where** part of the signature declaration. The interpretation of a signature t in its environment Υ is written $\{t_0 \mapsto \alpha_0, \ldots, t_n \mapsto \alpha_n\}(t)$ for type variables $t_0, \ldots, t_n$ and types $\alpha_0, \ldots, \alpha_n$ associated by Υ.

Definition 1 (Type expression semantics) For *a* term t from *STL*, the type denoted by t is written $\tau[\![t]\!]$ and defined as depicted in figure 5.3 with respect to the set of the following of predefined type-names: {**string**↦string, **character**↦character, **number**↦number, **boolean**↦boolean}.

Interpreting the first example service type in section 4.2 on page 44 is done in the environment

{**string**↦string, **character**↦character, **number**↦number, **boolean**↦boolean,
ccnumber↦string, date↦⟨number, number, number⟩$_A$, day↦number,
month↦number, year↦number, dest↦string, ack↦boolean, line↦string,
price↦⟨number,number⟩$_A$}

The interpretation results in the type

$$
\begin{aligned}
&\tau[\![\text{signature } \textbf{where} \text{ type-declarations}]\!] = \\
&\qquad \text{predefined} \cup [\![\text{type-declarations}]\!](\tau[\![\text{signature}]\!]) \\
&\tau[\![\text{operation-signature}_1 \text{ ; operation-signature}_2]\!] = \\
&\qquad \langle \tau[\![\text{operation-signature}_1]\!], \tau[\![\text{operation-signature}_2]\!] \rangle \\
&\tau[\![\text{operation-name : arguments} -\!> \text{results}]\!] = \\
&\qquad \text{operation-name} : \tau[\![\text{arguments}]\!] \to \tau[\![\text{results}]\!] \\
&[\![\text{type-declaration}_1 \text{ ; type-declaration}_2]\!] = \\
&\qquad [\![\text{type-declaration}_1]\!] \cup [\![\text{type-declaration}_2]\!] \\
&[\![\text{type-name} = \text{type-definition}]\!] = \{\text{type-name} \mapsto \tau[\![\text{type-definition}]\!]\} \\
&\tau[\![\text{type-definition}_1 \text{ * } \ldots \text{* type-definition}_n]\!] = \\
&\qquad \langle \tau[\![\text{type-definition}_1]\!], \ldots, \tau[\![\text{type-definition}_n]\!] \rangle_A \\
&\tau[\![\langle \text{ type-definition}_1, \ldots, \text{type-definition}_n \rangle]\!] = \\
&\qquad \langle \tau[\![\text{type-definition}_1]\!], \ldots, \tau[\![\text{type-definition}_n]\!] \rangle_A \\
&\tau[\![[\!] \text{ type-definition}_1, \ldots, \text{type-definition}_n [\!]]\!] = \\
&\qquad [\tau[\![\text{type-definition}_1]\!], \ldots, \tau[\![\text{type-definition}_n]\!]]_A
\end{aligned}
$$

Figure 5.3: Semantics of STL-expressions

$$
\begin{aligned}
&\langle \text{getflightticket: } \langle \text{string, } \langle \text{number, number, number} \rangle_A, \text{ string} \rangle_A \to \\
&\qquad \langle \text{boolean,} \langle \text{number,number} \rangle_A \rangle_A, \\
&\text{getbusticket: } \langle \text{string,} \langle \text{number,number,number} \rangle_A, \text{ string} \rangle_A \to \\
&\qquad \langle \text{boolean,} \langle \text{number,number} \rangle_A \text{,string} \rangle_A, \\
&\text{gettrainticket: } \langle \text{string,} \langle \text{number, number, number} \rangle_A, \text{ string} \rangle_A \to \\
&\qquad \langle \text{boolean,} \langle \text{number,number} \rangle_A \rangle_A \rangle.
\end{aligned}
$$

Let another booking-agency offer a service with the interface in figure 5.4. Interpreting

```
(getflightticket: ccnumber * date * dest -> ack * cashed;
 getbusticket   : ccnumber * date * dest -> ack * cashed;
 gettrainticket : ccnumber * date * dest -> ack * cashed)
where
 ccnumber       = string;
 date           = <number,number,number>;
 dest           = string;
 ack            = boolean;
 cashed         = <number,number>.
```

Figure 5.4: A service type for travel booking in STL

this definition results in the type

$$
\begin{aligned}
\langle \text{getflightticket: } & \langle \text{string, } \langle \text{number, number, number} \rangle_A, \text{ string} \rangle_A \rightarrow \\
& \langle \text{boolean,} \langle \text{number,number} \rangle_A \rangle_A, \\
\text{getbusticket: } & \langle \text{string,} \langle \text{number,number,number} \rangle_A, \text{ string} \rangle_A \rightarrow \\
& \langle \text{boolean,} \langle \text{number,number} \rangle_A \rangle_A, \\
\text{gettrainticket: } & \langle \text{string,} \langle \text{number, number, number} \rangle_A, \text{ string} \rangle_A \rightarrow \\
& \langle \text{boolean,} \langle \text{number,number} \rangle_A \rangle_A \rangle.
\end{aligned}
$$

Applying the subtyping rules on these types results in the judgement that the example type from section 4.2 on page 44 is a subtype of the one in figure 5.4.

In this section we defined a type system on which LAURA's STL-expressions are given semantics. The second part of this chapter deals with the behavior of LAURA's operations. However, we do not define it directly, but introduce a more abstract model of coordination with multisets which can be applied to any operations that are combinations of the addition and removal of some elements to and from a multiset. We exemplify this for LINDA and then finally define the behavior of LAURA's operations in this model.

5.2 A formal model of coordination: The Bag-Machine

In chapter 2 we described the coordination language LINDA, which coordinates agents in a parallel system using the tuple-space and four associated operations. In a similar way, ALICE and LAURA make use of a shared multiset of tuples and forms. In this section we define a machine and its behavior that forms a basis for uncoupled coordination in a LINDA-like fashion. It omits the details of specific coordination languages such as the outform of elements and will be used in the subsequent sections for a formal behavioral definition of the coordination languages LINDA and LAURA.
In a coordination language that uses a LINDA-like approach, three issues are of specific interest:

- Operations that deposit and withdraw elements to and from a shared multiset of elements,
- the structure of elements and means to construct and access them,
- a rule that guides the removal of an element with respect to a given pattern.

For LINDA, these issues are covered by the operations `out`/`eval` and `in`/`rd`, the tuples and associated constructors $\langle , \rangle$ together with binding-rules into a local environment, and finally the matching-rule given as a relation on tuples and templates. Similar, LAURA uses forms and another matching-relation and `service`/`serve`/`result` to put and withdraw forms to and from the service-space.

We can reason about operations on a multiset without having to detail out in what combination they form operations of a concrete coordination language. Here, we define a machine – the Bag-Machine – capable of performing operations on a multiset of elements shared by agents. We define rules for building terms of these operations enabling us to

subsequentially compose them into operations of concrete coordination languages. In the following we use the terms "bag" and "multiset" synonymously.

Let Bag-Machine be a machine that is able to perform operations on a multiset of elements with respect to some rule. Its operations are add, the deposition of an element in the bag and remove, the removal of an element from the bag that is in relation match to another element given as an argument. remove makes a nondeterministic choice if multiple elements of the bag match. If no matching element exists, the operation is delayed until one becomes available as the consequence of some add operation.

The operations of Bag-Machine shall be invoked by agents that are coordinated by using the shared multiset. These agents can be expected to work in a concurrent and distributed environment. Possible concurrent usages of Bag-Machine-operations include the following combinations:

1. add-operations can be performed concurrently. They do not interfere, even if elements of the same sort are added.

2. remove-operations can be performed concurrently if the sets of elements being in the match-relation with the given patterns for the operations are pairwise disjoint.

3. As many remove-operations resulting in elements of the same sort can be performed concurrently as how often this sort of element occurs in the bag.

4. add- and remove-operations on the same sort of element can be performed concurrently if there are as many occurrences of elements without those added as there are remove-operations executed.

In order to specify the effects of the operations of Bag-Machine, one would have to make statements about the number of occurrences of elements within the multiset of elements. Assuming that only one agent is using the Bag-Machine, the effect of add can be described as incrementing the number of occurrences of that element by one, that of remove as decrementing the number of occurrences of one matching element by one.

However, if we take case 4 above, and allow multiple agents to use the Bag-Machine concurrently, this number is undetermined for the single operations, as one agent adds an element and another performs remove concurrently on the same element, leaving the total number unchanged.

Determining the number of elements in the multiset requires some form of interleaving of operations so that at some time statements about the number of elements can be made without interference of concurrent operations. However, such an interleaving has the drawback to result in a total ordering of operations in which orderings caused by the semantics-preserving restrictions on concurrency are indistinguishable from those caused by interleaving. The ordering also reflects a choice on the atomicy of operations.

We argue that in the light of distributed systems, requiring a total order of operations is counter-intuitive and does neglect the benefits of distributed and concurrent systems which make them attractive. A distributed system does allow for truly concurrent and overlapping operations; it is desirable that nodes operate autonomously and a global state is avoided.

Guided by these observations, we decide to define the behavioral semantics of the Bag-Machine and of agents using it in terms of labeled event structures. Labeled event structures combine true concurrency with non-determinism allowing us to avoid the discussed problems.

Prior to the definition of event structures for operations on a multiset of elements by the Bag-Machine, we introduce a more detailed understanding of the multiset itself. Above, we listed access-combinations to the multiset that can take place concurrently. Let us recapitulate in detail to what our statements refer. Combination 1 refers to some elements, whether of the same sort or not. 2 refers to elements that are of distinct sorts. 3 refers to elements of the same sort, but makes a statement about different elements, as does 4.

We notice that there are two ways of reference to elements: As objects distinguished by their sort and as objects distinguished by their identity. In the following we speak of *elements* for those objects distinguished by their sort and *instances* for those distinguished by their identity.

The two ways of reference also can be found when talking about the matching mechanism and its behavior. Also, we notice different way of talking about object in the matching-rule and statements about concurrent behavior. A matching-rule is defined on elements and abstracts from instances. Dealing with behavior that makes the non-deterministic selections from the matching-rule concrete involves talking about instances.

The two ways of reference have counterparts in two mathematical notions of multisets discussed in [Monro, 87], multi-sets and multinumbers[1].

Let an example multiset of natural numbers consist of two instances of the element 10, called a and b and one instance of 20, called c. A *multinumber* is written as a set of elements that are labeled with numbers: $\{10^2, 20^1\}$. Let $\mathcal{E}$ be the set of all elements which can occur in a multiset ($\{10,20\}$ for the example). A multinumber then is a function $\mathcal{E} \to \mathbb{N}_0$, determining the labels in the above notation. Operations on multisets can be defined by using operations on natural numbers applied to the multinumber.

A second view can be taken by a *multi-set* which defines a multiset as a pair $\langle X_0, \varrho \rangle$, consisting of a set X_0 of instances and an equivalence relation ϱ on X_0. Our example would be notated as $\langle \{a, b, c\}, a \varrho b \rangle$, reflecting that a and b are considered to be of "the same sort" as they both are instances of 10.

Monro investigates in a category of multisets and gives definitions for multiset operations using category theory. Although a multi-set has an associated multinumber, not all operations for multi-sets are induced for multinumbers and vice versa.

We introduce a third view on multisets as certain objects in *ϵ-structures* ([Mahr and Sträter et al, 90], [Pooyan, 92]). The theory of ϵ-structures adopts the understanding that a set is given by the fact that statements about membership can be made. The statements of membership result in a relation amongst a set of objects, the carrier. Thus, an ϵ-structure is a pair $\mathcal{M} = (M, \epsilon)$, where M is the set of objects and ϵ a binary relation on M.

[1] In order to distinguish multisets in our discussion and their understanding in the referenced paper, we write "multi-set" to denote the special understanding of a multiset in contrast to that of a multinumber. Where the referenced paper uses the term multiset, we write multi-set.

We understand a multiset as being represented by a set of names for instances taken from an infinite set of names $\mathcal{N}$ – for the example this is $\{a, b, c\} \subseteq \mathcal{N}$. Each name stands for an instance of an element from the set of possible elements of the multiset $\mathcal{E}$ – $\mathbb{N}$ for the example. The set of names corresponds to X_0 from the multi-set and $\mathcal{E}$ to that of the multinumber.

The example multiset is represented as an ϵ-structure $\mathcal{M} = (M, \epsilon)$ with $M = \mathcal{N} \cup \mathcal{E} = \{a, b, c, 10, 20\}$ and $\epsilon = \{a\epsilon 10, b\epsilon 10, c\epsilon 20\}$. Figure 5.5 depicts the three views for another example multiset consisting of three instances of 2, called a, b and c, and two instances of 3, called d and e.

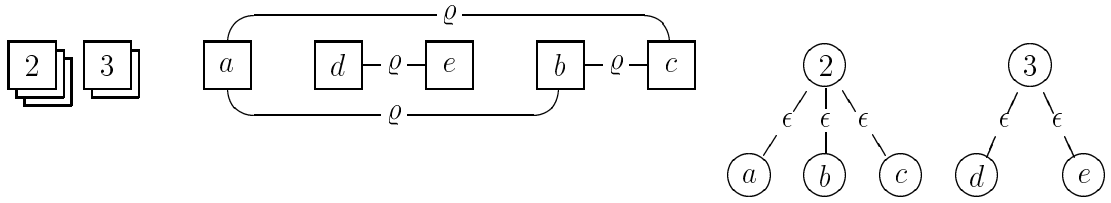


Figure 5.5: Different views on a multiset

Representing a multiset as an ϵ-structure captures both of the above views on multisets and allows us to integrate the element- and instance-level within a single framework. The add-operation then is the insertion of a relation between an instance and an element; remove is its removal. Given that the ϵ-structure is initially empty, add inserts both an element and an instance together with a relation between them if the element is not in the ϵ-structure. Otherwise only a new instance and a relation is inserted. remove removes an instance and the relation to an element. It also removes the element if there are no instances related to it. We now define the notations and mechanisms we will use later to define the behavior of Bag-Machine by giving a labeled event structure. Representing a multiset as an ϵ-structure captures both of the above views on multisets and allows us to integrate the element- and instance-level within a single framework. The add-operation then is the insertion of a relation between an instance and an element; remove is its removal. We now define the notations and mechanisms we will use later to define the behavior of Bag-Machine by giving a labeled event structure.

5.3 Technical preliminaries for the Bag-Machine

5.3.1 Labeled event structures

We follow [Winskel, 88] subsequentially and refer for the mathematical discussion of event structures to this source. Event structures provide us with true concurrent models of processes thus allowing us to follow our intent to give a non-interleaving behavioral specification.

An *event* is the occurrence of an action and can be localized in space and time. That is – as Winskel puts it –, any two events can be separated by some real number r that gives a radius in spheres of space and time. Modeling concurrent processes with event structures deals with constraints on the occurrence of events that reflect their dependencies, conflicts amongst them and – in consequence – their independence.

The relation $\leq$ on events expresses *causal dependency*, meaning that $e \leq e'$ if the occurrence of e' depends on the occurrence of e. $\mathrm{E} = (E, \leq)$ is called an *elementary event structure* and is a set E of events partially ordered by $\leq$. An elementary event structure usually has to satisfy the axiom of finite causes, that is $\forall e \in E : \{e' \in E | e' \leq e\}$ is finite.

An elementary event structure defines a partial order that imposes determinism. Dealing with non-determinism requires the ability to express that one out of many possible behaviors is chosen. This means that the occurrence of one event rules out the possibility of other events. This is captured by the notion of *conflict* amongst events, expressing that only one out of them can happen and that if it happens all others will not happen.

It is taken into account by a *prime event structure* $\mathrm{E} = (E, \#, \leq)$ consisting of a set E of events, partially ordered by $\leq$ augmented with the binary, symmetric and irreflexive conflict relation $\#$ on events. The conflict relation satisfies $e \# e' \leq e'' \Rightarrow e \# e''$ for $e, e', e'' \in E$.

A *stable event structure* $\mathrm{E} = (E, \#, \vdash)$ defines a more general model. Let Con be the set of those subsets $X \subseteq E$ that are *conflict-free*, i.e. $\forall e, e' \in X : \neg(e \# e')$. Then $\vdash \subseteq \mathrm{Con} \times E$ is the *enabling relation* which satisfies $(X \vdash e) \wedge (X \subseteq Y) \in \mathrm{Con} \Rightarrow Y \vdash e$. Thereby the occurrence of an event is not dependent on the occurrence of a single event but of sets of potential events.

In a *labeled event structure* $\mathrm{E} = (E, \#, \vdash, L, l)$ events are labeled from an alphabet of actions L by a label ling function $l : E \to L$. That is, an event is associated a label which describes the "nature" of the event – for example what action involves its occurrence.

To summarize: Labeled event structures relate events by causal dependency, by conflict and define unrelated events as independent. They induce a structure in which conflict free subsets define allowed occurrences of events. The crucial point is that all independent events can occur truly concurrent. We now introduce the sequential concatenation of two labeled event structures.

Definition 2 (Prefixing (following Winskel)) Let E be a labeled event structure consisting of $(E, \#, \vdash, L, l)$. Let a be a label. Define $a\mathrm{E}$ to be the labeled event structure $(E', \#', \vdash', L', l')$ with

$$
\begin{array}{rcl}
E' & = & \{(0, a)\} \cup \{(1, e) | e \in E\} \\
e'_0 \#' e'_1 & \Leftrightarrow & \exists e_0, e_1 : e'_0 = (1, e_0) \wedge e'_1 = (1, e_1) \wedge e_0 \# e_1 \\
X \vdash' e' & \Leftrightarrow & e' = (0, a) \vee (e' = (1, e_1) \wedge (0, a) \in X \wedge \{e | (1, e) \in X\} \vdash e_1) \\
L' & = & \{a\} \cup L \\
l'(e') & = & \begin{cases} a & , \text{if } e' = (0, a) \\ l(e) & , \text{if } e' = (1, e) \end{cases}
\end{array}
$$

Prefixing adds an event labeled a to the structure and changes the enabling relation so that the occurrence of all events depending on the occurrence of event $(0, a)$, which is always enabled. So, before anything can happen, $(0, a)$ has to happen. We extend this definition which prefixes an event-structure with a single event into a concatenation of two event-structures.

Definition 3 (Concatenation) Let $\mathrm{E}_0 = (E_0, \#_0, \vdash_0, L_0, l_0)$, $\mathrm{E}_1 = (E_1, \#_1, \vdash_1, L_1, l_1)$ be labeled event structures. Let a be a label. Define the *concatenation* $\mathrm{E}_0.\mathrm{E}_1$ to be the labeled event structure E'=$(E', \#', \vdash', L', l')$ with

$$
\begin{array}{rcl}
E' & = & \{(0,e)|e \in E_0\} \cup \{(1,e)|e \in E_1\} \cup \{(*,a)\} \\
e'_0 \#' e'_1 & \Leftrightarrow & (\exists e_0, e_1 : e'_0 = (0,e_0) \wedge e'_1 = (0,e_1) \wedge e_0 \#_0 e_1) \\
 & & \vee (\exists e_0, e_1 : e'_0 = (1,e_0) \wedge e'_1 = (1,e_1) \wedge e_0 \#_1 e_1) \\
X \vdash' e' & \Leftrightarrow & (e' = (0,e_0) \wedge \{e|(0,e) \in X\} \vdash_0 e_0) \\
 & & \vee (e' = (*,a) \wedge \{e|(0,e) \in X\} \in \mathrm{Con}_0) \\
 & & \vee (e' = (1,e_1) \wedge \{e|(1,e) \in X\} \vdash_1 e_1 \wedge (*,a) \in X) \\
L' & = & L_0 \cup L_1 \cup \{a\} \\
l'(e') & = & \begin{cases} l_0(e), \text{if } e' = (0,e) \\ l_1(e), \text{if } e' = (1,e) \\ a, \text{if } e' = (*,a) \end{cases}
\end{array}
$$

Concatenation of two event structures results in an event structure in which first a conflict free subset of events from the first event structure has to happen, after which a conflict free subset of events from the second can happen. This is achieved by introducing an event $(*, a)$ which depends on one of the conflict free subsets of E_0 and which is included in all enabling sets of events of E_1. $(*, a)$ thus can only happen after E_0 and anything from E_1 only after $(*, a)$. Figure 5.6 on page 66 depicts the concatenation of two identical event structures.

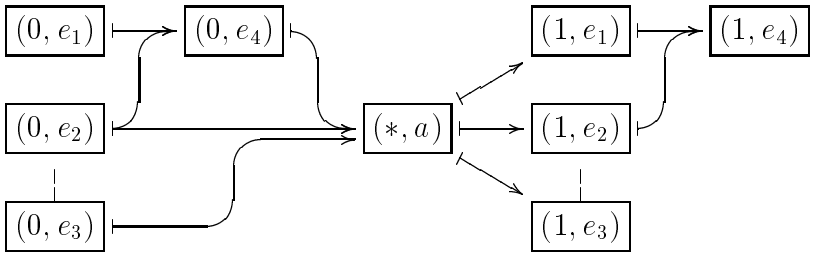


Figure 5.6: Concatenation of event structures

In our graphical representations we usually put the label of an event in a box, sometimes annotated with the event ($\boxed{label}^e$), use dashed lines (– – –) amongst events that are in conflict and arrows ($\mapsto$) to denote the enables-relation. Where the name of

an event is of more interest than its label – as in the example above –, we put the name of the event in the box. If an event is enabled by a set of events, these arrows join. For the sake of readability we mostly leave out enabled relations resulting from transitivity.

5.3.2 Open labeled event structures

We decided to use labeled event structures to define the behavior of the Bag-Machine because they provide a truly concurrent model. Communication of elements via the bag is asynchronous and without direct connections amongst agents. Such an uncoupled communication regards the deposition and removal of elements as distinct actions. Thus, it is not possible to model communication as a single event in an event structure, which is the approach in [Winskel, 88].

We now define *open labeled event structures* in which write and read actions are modeled as two distinct communication events. We classify events as follows:

Definition 4 (Classification) Let L be a set of labels and $R, W \subseteq L$ disjoint subsets of labels. Let $l(e) \in R$, if e is involved in an asynchronous read action and $l(e) \in W$ for an event that is involved in an asynchronous write action. Call R, W a *classification*.

The classification identifies labels of events that can be involved in some communication. They will be used in the parallel composition of event structures and are related by an associator:

Definition 5 (Associator) Define w to be a function on labels of events involved in read operations that results in a label a corresponding write operation has to have. We call w an *associator*.

An associator is a function $R \to W$ and identifies for a given label in R the label of an event that is to be involved in the same communication. With these, we can define open labeled event structures:

Definition 6 (Open labeled event structure) Let E be a labeled event structure, R, W a classification with $R \cup W \subseteq L$ and w an associator that is defined for all labels in R. We call $\mathrm{E}^{R,W,w}$ an *open labeled event structure* (OLES).

OLES are called open, as the labels that w results in are probably not used locally. The concatenation of OLES is defined as:

Definition 7 (Concatenation of OLES) The concatenation of OLES $\mathrm{E}_0^{R_0,W_0,w}$ and $\mathrm{E}_1^{R_1,W_1,w}$ with $W_0 \cap W_1 = \emptyset$ is defined to be $(\mathrm{E}_0.\mathrm{E}_1)^{R_0 \cup R_1, W_0 \cup W_1, w}$.

The concatenation does not change the classification as the additional events are not involved in communication but is defined only for event structures on the same associator. The parallel composition of OLES is defined as:

Definition 8 (Parallel composition of OLES) Let the OLES $\mathrm{E}_0^{R_0,W_0,w}$ be defined by $(E_0, \#_0, \vdash_0, L_0, l_0)$ and $\mathrm{E}_1^{R_1,W_1,w} = (E_1, \#_1, \vdash_1, L_1, l_1)$ with $W_0 \cap W_1 = \emptyset$. Their parallel composition $\mathrm{E}_0^{R_0,W_0,w} \| \mathrm{E}_1^{R_1,W_1,w}$ is defined as $\mathrm{E}^{R',W',w} = (E', \#', \vdash', L', l')$ with

$$
\begin{aligned}
E' &= \{(0,e) | e \in E_0\} \cup \{(1,e) | e \in E_1\} \\
e'_0 \#' e'_1 \Leftrightarrow &\quad (\exists e_0, e_1 : e'_0 = (0,e_0) \wedge e'_1 = (0,e_1) \wedge e_0 \#_0 e_1) \\
&\vee (\exists e_0, e_1 : e'_0 = (1,e_0) \wedge e'_1 = (1,e_1) \wedge e_0 \#_1 e_1) \\
&\vee (l(e'_0), l(e'_1) \in R \wedge l(e'_0) = l(e'_1)) \\
X \vdash' e' \Leftrightarrow &\quad (e' = (0,e_0) \wedge \{e | (0,e) \in X\} \vdash_0 e_0 \wedge \\
&\qquad \{(1,e) | l_1(e) \in W_1 \wedge l_1(e) = w(e')\} \in X) \\
&\vee (e' = (1,e_1) \wedge \{e | (1,e) \in X\} \vdash_1 e_1 \wedge \\
&\qquad \{(0,e) | l_0(e) \in W_0 \wedge l_0(e) = w(e')\} \in X) \\
L' &= L_0 \cup L_1 \\
l'(e') &= \begin{cases} l_0(e), \text{if } e' = (0,e) \\ l_1(e), \text{if } e' = (1,e) \end{cases} \\
R' &= R_0 \cup R_1 \\
W' &= W_0 \cup W_1
\end{aligned}
$$

The parallel composition results in an event structure in which events are in conflict if they are in conflict in the original structures or if they are read-events with identical labels. The enabling relation makes a read-event depending on a corresponding write-event from the other structure.

The composition operation relates the corresponding read- and write-events from both structures. The relations reflect which events can be involved in the same communication. Also, they reflect the uncoupled communication style as a write event enables multiple corresponding read events. However, these are in conflict, so conflict free subsets contain exactly one read event for one write event.

We will use the parallel composition of OLES to model the behavior of sets of agents that coordinate using the Bag-Machine. Figure 5.8(b) on page 72 contains an example of its application.

5.4 A labeled event structure for the Bag-Machine

We now can define the correct behavior of processes implementing the Bag-Machine by giving a labeled event structure $\mathrm{B} = (E, \#, \vdash, L, l)$. The events of interest E occur when executing add- and remove-operations.

Let the multiset on which the Bag-Machine works be represented as an ϵ-structure $\mathcal{M}$ as discussed above. It is defined as $\mathcal{M} = (M, \epsilon)$ where $M \subseteq \mathcal{N} \cup \mathcal{E}$ is the set of elements and instances representing the multiset and ϵ a relation $\mathcal{N} \times \mathcal{E}$ associating instances to elements.

The set of labels for the event structure is defined as a set of pairs of an operation name and a name, representing the identity of the object on which the operation is carried out. Thus, $L = \{(o, n) | o \in \{\mathsf{add}, \mathsf{remove}\}, n \in \mathcal{N}\}$.

The labeling function l labels events from E with labels from L so that an event e resulting in the execution of add and in the insertion of an ϵ-relationship for the instance named n in the ϵ-structure M gets the label (add, n) and an event f resulting in the execution of remove and in the removal of the ϵ-relationship for the instance named n in the ϵ-structure M is labeled (remove, n).

Furthermore, the labeling function has the property that $\forall e_0, e_1 \in E : l(e_0) = (\mathsf{add}, n) \wedge l(e_1) = (\mathsf{add}, m) \Rightarrow\ n \neq m$ for $e_0 \neq e_1$, that is no two events are involved in the insertion of equally named instances, or – equally – that instances have unique names[2]. With these, we define the conflict- and enabling-relations of B as

$$\begin{aligned} e_0 \# e_1 &\Leftrightarrow (l(e_0) = (\mathsf{remove}, n)) \wedge (l(e_1) = (\mathsf{remove}, n)) \\ X \vdash e &\Leftrightarrow X = \{e'\} \wedge (l(e') = (\mathsf{add}, n)) \wedge (l(e) = (\mathsf{remove}, n)) \end{aligned}$$

The conflict relation reflects that an instance can be removed only once. It also specifies the non-deterministic nature of element-removal in the bag machine. Let E be $\{e_0, e_1, e_2\}$ with $l(e_0) = (\mathsf{add}, n)$ and $l(e_1) = l(e_2) = (\mathsf{remove}, n)$. Then we have $\{e_0\} \vdash e_1$ and $\{e_0\} \vdash e_2$ reflecting that an instance can be removed by some event due to the uncoupled nature of the bag and the lack of addressing. However, since $e_1 \# e_2$, a non-deterministic choice has to be made on which events happens, i.e. which remove-operation involving e_1 or e_2 results in the instance named n.

The enabling relation reflects that a remove-operation on a specific instance depends on an add-operation on this instance. As the enabling set of events has – by the definition of l – only one element, the enabling relation is equal to causal dependency as in a prime event structure.

It seems possible to make the labeling function less constrained regarding the (add, n)-labels and to extend the conflict relation # by events (e_0, e_1) with $(l(e_0) = (\mathsf{add}, n)) \wedge (l(e_1) = (\mathsf{add}, n))$. This, however would – by the inheritance of # from $\leq$ – bring events labeled (add, n) and (remove, n) in conflict.

Let us return to the access patterns we enumerated in section 5.2 and check if the event structure B identifies them as independent and thus concurrent and potentially overlapping. For case 1, the concurrent add-operations, they involve a set of events $\{e_i, \ldots, e_k\}$ which are labeled by $(\mathsf{add}, n), \ldots, (\mathsf{add}, o)$. These events are identified as independent. Case 2 involves events $\{e_i, \ldots, e_k\}$ labeled $(\mathsf{remove}, n), \ldots, (\mathsf{remove}, o)$. These are identified as independent as the labels differ because they operate on instance of different elements, so the names have to be unequal. For 3, two sets of events are involved: $\{e_i, \ldots, e_k\}$ and $\{e_l, \ldots, e_o\}$, which are labeled $(\mathsf{add}, n), \ldots, (\mathsf{add}, o)$ and $(\mathsf{remove}, n), \ldots, (\mathsf{remove}, o)$. Here, the enabling-relation constrains all remove-operations to be enabled by corresponding add-operations on the same name but imposes no restriction on the concurrent occurrence of the remove-events. Finally, case 4 is a combination of cases 1 and 3.

[2] In the following, we therefore sometimes use the terms instances, names and names of instances synonymously.

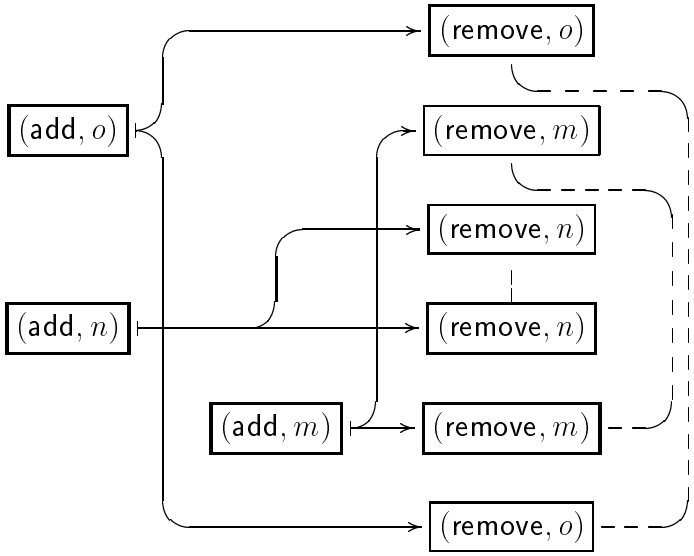


Figure 5.7: An event structure for the bag-machine

Figure 5.7 depicts a set of events for three add- and remove-operations including conflict and enabled relations. The events having a thicker border form one conflict-free subset of B, and can be identified as correct behavior of the Bag-Machine.

The definition of the behavior of Bag-Machine in terms of an event structure does not impose unnecessary restrictions on an implementation of the Bag-Machine. It turned out that the outlined understanding of multisets is enabling for this solution. However, the event structure specifies only the behavior of processes implementing the Bag-Machine. In the next section we define the behavior of terms of operations on the Bag-Machine and that of a system of agents.

5.5 The behavior of agents using the Bag-Machine

So far, we spoke of "some agents acting on the Bag-Machine concurrently." The event structure in the previous section considered only the process implementing the Bag-Machine. To define coordination languages that use the Bag-Machine as the underlying synchronization and communication mechanism, we will form terms of operations and consider systems of agents working concurrently. In this section we introduce operators for generating terms and a definition to describe a system of concurrent agents. When defining a language based on these operators, their behavioral semantics is given by the definitions here.

Agent-terms will be expressed in a language L with the syntax

$$p ::= halt \mid add(e).p \mid remove(e).p \mid local.p$$

halt identifies the end of an agent-term and denotes its termination. Most of the time this will not be explicitly expressed. *add* means the addition of some element and *remove* the removal of some element using the Bag-Machine. *local* denotes some local operation which has no external effects and corresponds to computation.

A system of agents working concurrently using the Bag-Machine for uncoupled communication and synchronization has the form

$$p_0 \| p_1 \| \ldots \| p_n$$

Here, $\|$ is the parallel composition of agents that execute some process given by a term and synchronize and communicate using the Bag-Machine only. We will give semantics to terms and systems of agents by open labeled event structures as defined in section 5.3.2.

Events involved in read- and write-actions are always labeled with pairs (o, n) with $o \in \{\mathsf{add}, \mathsf{remove}\}$ and $n \in \mathcal{N}$ as described in the previous section. Thus, the labels from some set L can be classified by mapping $\mathcal{R}_B(L) = \{(\mathsf{remove}, n) | n \in \mathcal{N} \wedge (\mathsf{remove}, n) \in L\}$, the labels of events that are involved in the reading of elements from the bag and $\mathcal{W}_B(L) = \{(\mathsf{add}, n) | n \in \mathcal{N} \wedge (\mathsf{add}, n) \in L\}$, involved in the writing of elements. An associator w_B can be given by $w_B((\mathsf{remove}, n)) = (\mathsf{add}, n)$ for some n from $\mathcal{N}$.

Definition 9 (Denotational semantics) For a term t from L, the denotation of t is written $[\![t]\!]$ and defined as follows with respect to $\mathcal{R}_B, \mathcal{W}_B$ and w_B

$$\begin{aligned}
[\![halt]\!] &= (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)^{\emptyset, \emptyset, w_B} \\
[\![add]\!] &= (\{e\}, \emptyset, \emptyset, (\mathsf{add}, n), l(e) = (\mathsf{add}, n))^{\emptyset, \{(\mathsf{add}, n)\}, w_B} \\
[\![remove]\!] &= (\{e\}, \emptyset, \emptyset, (\mathsf{remove}, n), l(e) = (\mathsf{remove}, n))^{\{(\mathsf{remove}, n)\}, \emptyset, w_B} \\
[\![local]\!] &= local^{\emptyset, \emptyset, w_B} \\
[\![t_1.t_2]\!] &= [\![t_1]\!]^{R_B, W_B, w_B}.[\![t_2]\!]^{R_B, W_B, w_B} \\
[\![t_1 \| t_2]\!] &= [\![t_1]\!]^{R_B, W_B, w_B} \| [\![t_2]\!]^{R_B, W_B, w_B}
\end{aligned}$$

That is, *halt* denotes an empty OLES which contains no events and an empty classification. *add* denotes an event structure involving an event e which is uniquely labeled (add, n) and classified as writing. The event structure denoted by *remove* is similar with $l(e)$ being classified as reading. The semantics of *local* is left open as it depends on the computation language the coordination language is combined with. We know, however, that it involves no coordination, thus no labels are classified as reading or writing. The sequential composition of terms denotes event structures generated by our concatenation operation. A set of agents executing terms is the parallel composition of the open labeled event structures denoted by the terms.

Our open labeled event-structures and L offer no choice-operation or choice-construct, as one probably would expect. In fact, the choice is not necessary for our purposes.

In section 3.3.1 we demonstrated for ALICE, how if-then-else constructs can be implemented in a coordination languages that has no control structures except for sequencing. The mechanism was to encode the condition and the two branches in a tuple and to have agents waiting that start one branch-process depending on the condition.

One of them was selected by matching the condition value with a constant true or false. Thus, the choice of which branch to take is made by the matching-mechanism. Therefore, we do not introduce a choice-operation here.

As an example for an open labeled event-structure for a system of agents, figure 5.8(a) on page 72 shows the event-structures denoted by the terms remove(e), add(e).remove(f) and add(f).remove(e). The events a_1 and a_2 are those introduced by the concatenation. In figure 5.8(b) they are composed into a system of agents with appropriate enabling and conflict relations.

If we now remove all events and relations that are either local or result from sequencing within the processes, the event-structure in figure 5.8(c) results. In fact, this event-structure corresponds to the event-structure we illustrated in figure 5.7 for the process that implements the Bag-Machine.

We can formalize this extraction of the Bag-Machine process from a system of agents by the following coord-operation on open labeled event-structures that use the Bag-Machine for coordination:

Definition 10 (Coordination structure) Let E be an open labeled event structure $(E, \#, \vdash, L, l)^{R_B, R_B, w_B}$. The coordination structure then is the labeled event structure E'=$(E', \#', \vdash', L', l')$, written coord(E) with

$$\begin{aligned} E' &= \{e | l(e) \in R_B \vee l(e) \in W_B\} \\ e'_0 \#' e'_1 &\Leftrightarrow e'_0 \in E' \wedge e'_1 \in E' \wedge e_0 \# e_1 \\ X \vdash' e' &\Leftrightarrow \forall e \in X : e \in E' \wedge e' \in E' \\ L' &= \{l | l \in R_B \vee l \in W_B\} \\ l'(e') &= l(e) \end{aligned}$$

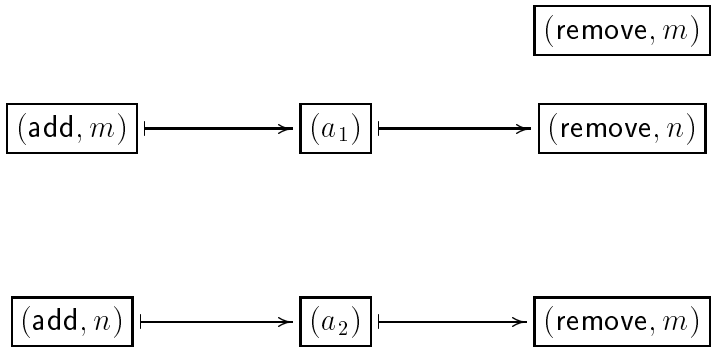
It turns out that coord($[\![t_1 \| \ldots \| t_n]\!]$)=B, the event-structure for the Bag-Machine as defined in section 5.4:

Theorem 1 With respect to R_B, W_B, w_B, a coordination structure of the open event structure denoted by the parallel composition of terms $t_1, \ldots, t_n$ equals to the event structure B for the Bag-Machine process, that is coord($[\![t_1 \| \ldots \| t_n]\!]$)=B.

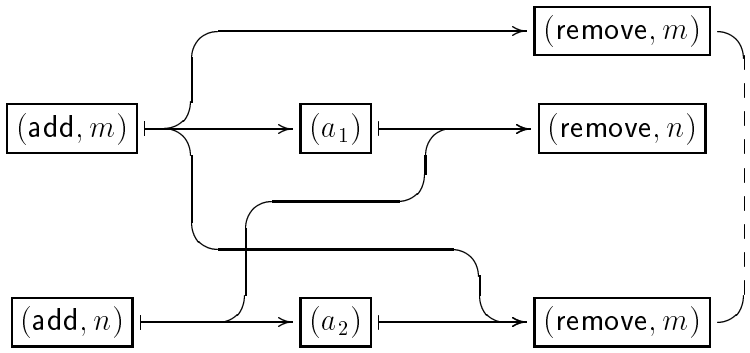
Proof 1 From definition 9, events in $[\![t_1 \| \ldots \| t_n]\!]$ are labeled (add,n) or (remove,n) with $n \in \mathcal{N}$ or are labeled according to *local*. Also from definition 9, R_B is a set $\{(\mathsf{remove}, n) | n \in \mathcal{N}\}$ and W_B a set $\{(\mathsf{add}, n) | n \in \mathcal{N}\}$.

From definition 10, the set of events in coord($[\![t_1 \| \ldots \| t_n]\!]$) is a set E with $\forall e \in E : (l(e) = (\mathsf{add}, n), n \in \mathcal{N}) \vee (l(e) = (\mathsf{remove}, n), n \in \mathcal{N})$. These events are the only ones that occur in B. The set of labels in coord($[\![t_1 \| \ldots \| t_n]\!]$) is L with $\forall l \in L : (l = (\mathsf{add}, n), n \in \mathcal{N}) \vee (l = (\mathsf{remove}, n), n \in \mathcal{N})$. This equals to the set of labels $\{(o, n) | o \in \{\mathsf{add}, \mathsf{remove}\}, n \in \mathcal{N}\}$ defined for B.

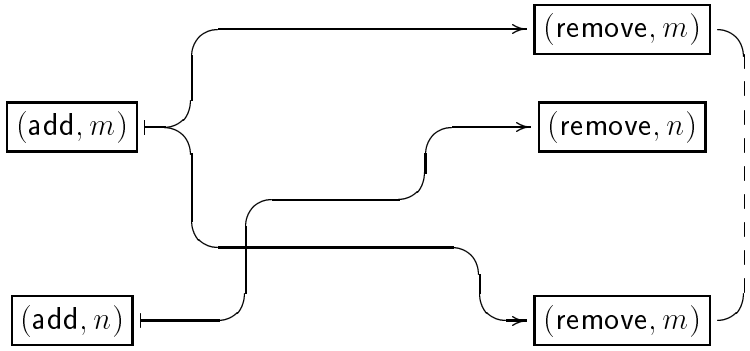
From definition 9 we have that an event e involved in $[\![\mathrm{add}]\!]$ is labeled (add, n) and an event e involved in $[\![\mathrm{remove}]\!]$ is labeled (remove, n). Therefore, the labeling function in coord($[\![t_1 \| \ldots \| t_n]\!]$) labels event according to the labeling function for B.



(a) Uncoordinated agents: $[\![a_1]\!]$, $[\![a_2]\!]$, $[\![a_3]\!]$



(b) Coordinated agents: $[\![a_1 \| a_2 \| a_3]\!]$



(c) The Bag-Machine coordinates: coord($[\![a_1 \| a_2 \| a_3]\!]$)

Figure 5.8: The Bag-Machine process is embedded in a system of agents

From definition 8 events e_0, e_1 in $[\![t_1 \| \ldots \| t_n]\!]$ are in conflict if $(l(e_0), l(e_1) \in R_B \wedge l(e_0) = l(e_1))$. As R_B is $\{(\mathsf{remove}, n) | n \in \mathcal{N}\}$, this equals to the conflict relation for B. There are no other events that are in conflict in coord($[\![t_1 \| \ldots \| t_n]\!]$).

From definition 8, an event e is enabled by the union of the events that enable it in one OLES plus events are are classified as writing and whose labels are equal to the

associated write event for e. For a coordination structure with respect to W_B and w_B, this means that an event e_1 from E_1 enables an event e, if its label is in $\{(\mathsf{add}, n)|n \in \mathcal{N}\}$ and the label of e is (remove, n). As the coordination structure contains only events labeled (remove, n) or (add, n), the enabling relation in the coordination structure is $X \vdash e' \Leftrightarrow \{e|l(e) \in \{(\mathsf{add}, n)|n \in \mathcal{N}\} \wedge l(e) = w((\mathsf{remove}, n))\} \in X$. Therefore $l(e') = (\mathsf{remove}, n)$ and as the events involved in $[\![add]\!]$ following definition 9 are labeled uniquely, there is only one event fulfilling this condition. Therefore, the enabling relation equals to the one defined for B.

With these definitions, we are ready to use terms of operations on the Bag-Machine to define operations of coordination languages that base on multisets. We will do so in section 5.7 for a subset of LINDA but first an interface for the embeddings with computation languages has to be defined.

5.6 Embedding coordination and computation languages

Coordination languages focus on coordination of agents whereas computation languages are intended for the expression of the actual work the agents perform. The use of a coordination language requires an embedding with a computation languages. Specifying a coordination language has to take this into account by being generic towards one or more computation languages. In the event structure above, we abstracted from all computation by the *local*-operation. However, for the definition of a coordination language, we have to give more details.

In this section we specify the minimal interface that is needed for an embedding. It includes the specification of types and values of a language, an abstraction from expressions and their evaluation and the definition of mappings between types and values of different languages. The latter is necessary as the abstraction from programming languages in our coordination approach implies the possibility of the use of multiple computation languages for agents. Communication amongst them requires the mapping of their types and values.

We can assume that a computation language has some type system that gives a structure to the set of values on which computations can be performed. For the issue of coordination, the sorts Type and Value are of interest, as typed values are communicated. They are captured in the signature typesystem below. The notations we use here match those defined in [Ehrig and Mahr, 85]. Also, definitions of some basic types such as Nat or Bool can be found there.

Throughout this chapter we use this notation to define signatures and data types. The equation we give do, however, not define mathematically sound semantics for the data types introduced. It is our intention – as stated in the introduction to this chapter – to bridge the gap between the informal description of LAURA and implementations of the system by formalizing the most important components of it. Thus the equations given in the following are requirements that an implementation of LAURA has to fulfill. We do not intend to define a "LAURA-calculus" which can be used for theoretical insights.

As put above, we expect agents to be written in multiple languages, which makes it necessary to have mappings amongst values and types from different type systems. In order to avoid name-conflicts, external defines a renaming of a typesystem.

typesystem = Bool +
sorts
  Type, Value
operations
  isof: Value × Type → Bool

external(ts) =
  ts **renamed** by
  ExtType **for** Type
  ExtValue **for** Value
  extisof **for** isof

Figure 5.9: A signature for a type system and its renaming

We can understand computation as the evaluation of expressions resulting in some values. The complexity of expressions depends on the computation language – it can be an arithmetic expression, a procedure expression or a complete program. We are not interested in the details of an evaluation, however, we want to formulate that an expression could be evaluated concurrently to other computations.

Furthermore, a computation language can be assumed to have some form of binding-concept. That is, we assume that an agent has a local environment that stores data from which values can be referenced within expressions and to which new values can be written. These assumptions are captured in the signature language below, which is parameterized with some type system.

language(ts) = ts + Bool +
sorts
  Expression, Binding
operations
  evaluate: Expression → Value
  spawn: Expression → Value
  isof: Binding × Type → Bool
  bound: Binding × Value → Bool

Figure 5.10: A signature for a computation language

Here, evaluate is the evaluation of some Expression resulting in a value, as is spawn. The concept of binding involves some form of binding-rules Binding – in a language like C such a binding-rule is the left hand side of an assignment. A binding-rule expects some type of value to be bound; isof reflects this. The effect of a binding is captured by the bound-predicate, being true if a value is bound according to a binding-rule.

Now we are ready to define a data type that constitutes an interface for a computation language to be embedded with a coordination language. It is defined below as embedding.

embedding(host,ext) =
  language(host) + external(ext) +
operations
  externalize: Type → ExtType
  externalize: Value → ExtValue
  internalize: ExtType → Type
  internalize: ExtValue → Value
equations
  t ∈ Type, v ∈ Value, u ∈ ExtType, w ∈ ExtValue
  internalize(externalize(t)) = t
  externalize(internalize(u)) = u
  internalize(externalize(v)) = v
  externalize(internalize(w)) = w

Figure 5.11: A data type for an embedding

embedding is parameterized with two languages, the host-language for the embedding and some external language. The operations internalize and externalize are mapping operations between the values and types of the host-languages to those of the external type system.

We can identify three situations that have different impacts on the outform of these mappings. Say we want to coordinate agents written in the language C only, then we need one embedding embedding(C,C) and the mappings are identity functions. If we have agents written in the languages C, Pascal and ML and chose Pascal's type- and value system for the external representations then we have embeddings based on embedding(C,Pascal), embedding(Pascal,Pascal), and embedding(ML,Pascal), where mappings from C and ML to Pascal have to be provided. Finally, we can have some external value- and type system, say XDR ([SUNa], [SUNb]), requiring embeddings embedding(C,XDR), embedding(Pascal,XDR), and embedding(ML,XDR). If for some language such mappings are impossible, agents written in that language are not suited to be coordinated.

With an embedding and the Bag-Machine operations, we can formulate a specification of LINDA and its embedding in some computation language. This is the topic of the next section.

5.7 Example: Specifying LINDA with the Bag-Machine

The specification of LINDA with Bag-Machine consists of two data-types. The first defines the elements of the tuple-space, tuples and templates, the second the LINDA-operations as terms of Bag-Machine-operations.

Figure 5.12 shows the specification of the elements that are used for coordination with LINDA. They are parameterized with some host-language and some external type system. LINDA distinguishes templates and tuples, but uses the same ⟨⟩-constructor for both. We specify ⟨⟩ for tuples built from a list of actuals and ⟨⟩' for templates built

Linda-elements(host,ext) = embedding(host,ext) + Bool + Nat +

sorts

Tuple, Template, Field, Actual

operations

$\langle \_ \rangle$: Actual* $\rightarrow$ Tuple

$\langle \_ \rangle'$: Field* $\rightarrow$ Template

length: Tuple $\rightarrow$ $\mathbb{N}$

length: Template $\rightarrow$ $\mathbb{N}$

fieldn: Tuple $\times$ $\mathbb{N}$ $\rightarrow$ Field

fieldn: Template $\times$ $\mathbb{N}$ $\rightarrow$ Field

field: Formal $\rightarrow$ Field

field: Actual $\rightarrow$ Field

actual: Value $\rightarrow$ Actual

actual: Expression $\rightarrow$ Actual

formal: Type $\rightarrow$ Formal

formal: Binding $\rightarrow$ Formal

tuplebound: Template $\times$ Tuple $\rightarrow$ Bool

fieldbound: Field $\times$ Field $\rightarrow$ Bool

match: Template $\times$ Tuple $\rightarrow$ Bool

matchfield: Field $\times$ Field $\rightarrow$ Bool

equations

$v \in$ Value, $tp \in$ Type, $e \in$ Expression, $b \in$ Binding

$t \in$ Tuple, $u \in$ Template, $f \in$ Field, $a, a_1, \ldots, a_n \in$ Actual

$xv \in$ ExtValue, $xt \in$ ExtType

length($\langle a_1,\ldots,a_n \rangle$) = n

length($\langle f_1,\ldots,f_n \rangle'$) = n

fieldn($\langle a_1,\ldots,a_i,a_j,a_k,\ldots,a_n \rangle$,j) = field($a_j$)

fieldn($\langle f_1,\ldots,f_i,f_j,f_k,\ldots,f_n \rangle'$,j) = f_j

actual(v) = externalize(v)

actual(tp) = externalize(tp)

actual(e) = actual(evaluate(e))

formal(b) = formal(typeof(b))

tuplebound(u,t) = $\bigwedge_{i=1}^{\text{length(t)}}$ fieldbound(fieldn(u,i),fieldn(t,i))

fieldbound(field(xv),field(xv)) = TRUE

fieldbound(field(formal(tp)),field(xv)) = TRUE

fieldbound(field(formal(b)),field(actual(xv))) = bound(b,internalize(xv))

match(u,t) = $\bigwedge_{i=1}^{\text{length(t)}}$ matchfield(fieldn(u,i),fieldn(t,i)) $\wedge$ (length(u) = length(t))

matchfield(field(xv),field(xv)) = TRUE

matchfield(field(xt),field(xv)) = TRUE, if extisof(xv,xt)

Figure 5.12: LINDA's tuples as an abstract data type

from a list of fields, including formals. length and the projection fieldn yield the length of a tuple or template or one of their fields resp.

Actuals are generated from values being mapped to the external representation, as are formals from a type. An actual can also result from the evaluation of an expression and a formal can be constructed from a binding-rule by determining a type expected in the rule.

The communication of a tuple results in the binding of its contents to the environment of an agent by binding its fields according to the template. This is reflected in the tuplebound and fieldbound predicates. If both fields are values or if the template-field contains only a type, then the values do not have to be bound. If the template field contains a binding-rule, the field is bound if the internalized field from the tuple is bound according the binding-rule.

Matching of a template and a tuple requires that all fields match and that the template and the tuple are of the same length. This is reflected by the match and matchfield predicates. Fields match, if they contain the same value or if the value of the tuple-field is of the type requested in the template-field.

Note that matching is performed on types and values in the external type system. The externalization takes place with actual and formal, the internalization with fieldbound.

The second part of the LINDA-specification consists of the definition of LINDA's operations in terms of Bag-Machine. We define LINDA's operations in terms of how their effects are visible to an agent. An agent performing an `in` observes the effect of having a tuple returned that matches the given template. The agent does not know about the tuple-space, about other agents acting concurrently on the tuple-space or about the time when the matched tuple was inserted into the tuple-space.

"Behind the stages" there is a tuple-space and other agents. And there is some mechanism that ensures a correct behavior of the tuple-space. There is some mechanism that makes non-deterministic choices on matching tuples. The entity acting behind the stages is the Bag-Machine whose behavior was defined by an event structure in section ??. To agents, the Bag-Machine is visible as an interface of a library. In order to use this interface in the following, we define it as a signature as follows, while leaving open its the concrete semantics:

"Behind the stages" there is a tuple-space and other agents. And there is some mechanism that ensures a correct behavior of the tuple-space. There is some mechanism that makes non-deterministic choices on matching tuples. The entity acting behind the stages is the Bag-Machine whose behavior was defined by an event structure in section 5.4. To use the Bag-Machine in the framework of data types, bag-machine makes its operations available to agents by the following signature.

bag-machine(Element) =
operations
add: Element → Element
remove: Element → Element

Figure 5.13: Signature for agents using the Bag-machine

add performs the add-operation and returns the element unchanged. remove performs the remove-operation and results in an element that is in the match-relation with the argument element. The semantics of the application of operations from Bag-machine is given by translating them into terms of L in section 5.5. That is $[\![\text{remove(s)}\cdot\text{f}\cdot\text{add(t)}]\!] = \text{add(t).local.remove(s)}$. With these, we can define LINDA's operations as they are made available to agents:

Linda(host,ext) = Linda-elements(host,ext) + bag-machine(Template) +
operations
  out: Tuple → Tuple
  eval: Expression* → Tuple
  in: Template → Tuple
  rd: Template → Tuple
equations
  t ∈ Tuple, u ∈ Template
  $e_1, \ldots, e_n$ ∈ Expression
  out(t) = add(t)
  eval(ts,t) = out(⟨spawn(e_1),...,spawn(e_n)⟩)
  in(u) = t, t=remove(u), iff tuplebound(u,t)
  rd(u) = add(in(u))

Figure 5.14: LINDA's operations

"Behind the stages" of LINDA, a Bag-Machine performs coordination on a multiset of templates with appropriate matching. out is the addition of a tuple to the multiset. eval takes a list of expressions, which are evaluated concurrently into values. As with the original LINDA-literature, we do not specify the order of evaluation and the synchronization of the collection of results. The results are taken to form a tuple which is added to the multiset via out. In order not to complicate the specification, we can regard constant values as expressions without restricting eval.

in equals to the removal of an element from the multiset, if the fields of the tuple are bound to the local environment of the agent according to the binding rules in the template. rd can be specified as the combination of in and the subsequential out of the result. The non-deterministic nature of `in` and `rd` allow the temporary withdrawal of a tuple, as no agent is guaranteed to access a tuple that is "under inspection" by a `rd` of another agent.

In the same style as we specified LINDA as an example, we define LAURA in the next section.

5.8 The semantics of LAURA's operations

We give a formal specification in the style we demonstrated in the previous section for LINDA. First, we start with LAURA's type system defined in figure 5.15. lauratypes is the external type system for which an embedding has to provide mappings to and from types and values.

Lauratypes = Bool+Real +
sorts
Type, Value, Servicetype, Opsig, Opcode, Uid
operations
character: → Type
string: → Type
boolean: → Type
number: → Type
'␣'...'~': → Value
{'␣'...'~'}*: → Value
FALSE, TRUE: → Value
IR: → Value
⟨_,...,_⟩: Type* → Type
[_,...,_]: Type* → Type
_→_: Type × Type → Type
_:_→_: Opcode × Type × Type → Opsig
(_*): Opsig* → Servicetype
isof: Value × Type → Bool
equations
v ∈ Value
v ∈ {'␣'...'~'} ⇔ isof(v,character)
v ∈ {'␣'...'~'}* ⇔ isof(v,string)
v ∈ {'FALSE','TRUE'} ⇔ isof(v,boolean)
v ∈ IR ⇔ isof(v,number)

Figure 5.15: A specification of LAURA's type system

It defines the four basic types character, string, boolean and number along with constant symbols that generate values of these types. Typing of the constants is defined by the properties of the isof-relation.

The constructors for records, unions and functions generate complex types. Note that a mapping of complex types to and from some host language does not necessarily require the mapping of the complex types found in that language. It suffices to provide some syntactical means to construct them.

The sorts Servicetype, Uid and Opcode are used in forms. Their representations are not "visible" for a host language and need not to be mapped.

The representation of values and types depends on the implementation of the type system. In the experimental implementation in the next chapter for example, we use XDR to represent values and character strings to represent service types and opcodes. Given this external type system, we can specify LAURA's forms as shown in figure 5.16 and figure 5.17.

Laura-forms(host) = embedding(host,Lauratypes) + Bool + Nat +
sorts
Form, UForm
operations
$\langle \_ \rangle$: ServiceType × Opcode × Actual* × Formal* → Form
putform: Servicetype × Opcode × Actual* → Form
getform: Servicetype × Opcode × Formal* → Form
serveform: Servicetype × Formal × Formal* → Form
resultform: Servicetype × Opcode × Actual* → Form

servicetype: Form → ServiceType
opcode: Form → Opcode
args: Form → Field*
res: Form → Field*

unique: Form → UForm
uniquify: Form × Uid → UForm
strip: UForm → Form

length: Field* → IN
fieldn: Field* × IN → Field
field: Formal → Field
field: Actual → Field
actual: Value → Actual
actual: Expression → Actual
actual: Opcode → Actual
formal: Type → Formal
formal: Binding → Formal
formal: Opcode → Formal

argsbound: Form × Form → Bool
resbound: Form × Form → Bool
fieldbound: Form × Form → Bool

match: UForm × UForm → Bool
equations
... (continued in figure 5.17)

Figure 5.16: Specification of LAURA's forms (part 1)

Laura-forms(host) = embedding(host,Lauratypes) + Bool + Nat +
sorts
Form, UForm
operations
... : *(continued from figure 5.16)*
equations
s, t ∈ Servicetype, o, p ∈ Opcode, a, b ∈ Actual*, q, r ∈ Formal*
v ∈ Value, tp ∈ Type, bi ∈ Binding, xv ∈ ExtValue
f, g ∈ Form, u, w ∈ Uid, uf ∈ UForm
putform(s,o,a) = ⟨s,o,a,⟩
getform(s,o,r) = ⟨s,o,,r⟩
serveform(s,formal(o),r) = ⟨s,formal(o),r,⟩
resultform(s,o,a) = ⟨s,o,,a⟩
servicetype(⟨s,o,a,r⟩) = s
opcode(⟨s,o,a,r⟩) = t
args(⟨s,o,a,r⟩) = a
res(⟨s,o,a,r⟩) = r
strip(uniquify(f,u)) = u
length(a_1,...,a_n) = n
fieldn(a_1,...,a_i,a_j,a_k,...,a_n,j) = field(a_j)
actual(v) = externalize(v)
actual(tp) = externalize(tp)
formal(bi) = formal(typeof(bi))
argsbound(f,g) = $\bigwedge_{i=1}^{\text{length(args(g))}}$ fieldbound(fieldn(args(f),i),fieldn(args(g),i))
resbound(f,g) = $\bigwedge_{i=1}^{\text{length(res(g))}}$ fieldbound(fieldn(res(f),i),fieldn(res(g),i))
fieldbound(field(xv),field(xv)) = TRUE
fieldbound(field(formal(tp)),field(xv)) = TRUE
fieldbound(field(formal(b)),field(actual(xv))) = bound(b,internalize(xv))
match(unique(getform(s,o,r)),unique(getform(t,p,q))) = FALSE
match(unique(getform(s,o,r)),unique(putform(t,p,b))) = FALSE
match(unique(getform(s,o,r)),unique(serveform(t,p,q))) = FALSE
match(uniquify(getform(s,o,r),u),uniquify(resultform(t,p,b),u)) = TRUE
match(unique(putform(s,o,a)),unique(putform(t,p,b))) = FALSE
match(unique(putform(s,o,a)),unique(resultform(t,p,b))) = FALSE
match(unique(putform(s,o,a)),unique(serveform(t,p,q))) = TRUE, iff t⩽ s
match(unique(serveform(s,o,r)),unique(serveform(t,p,q))) = FALSE
match(unique(serveform(s,o,r)),unique(resultform(t,p,b))) = FALSE
match(unique(resultform(s,o,a)),unique(resultform(t,p,b))) = FALSE
match(u,w) = match(w,u)

Figure 5.17: Specification of LAURA's forms (part 2)

A Form contains four major parts: a Servicetype indicating the service requested or offered, an Opcode denoting the operation requested, a list of Actuals as argument values and a list of Formals as placeholder for the results.

There are four specific kinds of forms. A service-put form is generated by putform with the placeholder for results omitted, while a service-get form is generated by getform

with the arguments omitted. Both are used for `service` – a service-put form to put the service-request into the service-space, a service-get form to get the results from the service-space.

A serve-form generated by serveform is used for `serve` where only placeholder for the arguments are used. `result` requires a result-form generated by resultform. With projections servicetype, opcode, args and res the four components of a Form can be accessed.

The form-transformations described above transform a Form into a unique form of sort UForm with unique and uniquify. A unique form is transformed into a Form by strip.

Similar to the specification of LINDA above, length, fieldn, field, actual and formal specify how lists of fields are constructed and accessed. The communication aspect is modeled by the predicates argsbound and resbound. They hold, if the values from an argument- or result-list of a form are bound to the local environment of an agent according to the binding-rules found in a second form. Note, that the externalization and internalization of fields is again captured with actual and fieldbound.

Finally, the match-predicate is defined for unique forms. Given the four kinds of forms, only the pairs service-put/serve and service-get/result can match.

A service-put form matches a serve-form, if the offered servicetype is a subtype of the one requested. This subtype relation is to be inferred from the semantics of the two service interfaces given in figure 5.3 and the equivalence- and subtyping-rules for servicetypes in figures 5.1 and 5.2.

A unique result-form and a unique service-get form match, if they contain the same unique identifier. In both cases, we assume that the embedding ensures that the argument- and result-lists contain values and placeholder that are typed consistent with the typing of the requested operation.

With the specification of the forms, we can formalize LAURA's operations as shown in figure 5.18. Laura is parameterized with some host-language and makes use of a bag-machine that operates on a multiset of unique forms.

The effect of service observable for an agent is that of taking a form with a service request and filling out the results. It is achieved by adding a unique form via service-put to the service-space with add and by receiving it via service-get from there. The retrieval with remove has the side effect of binding the result-values to the environment of the agent.

serve takes a service-offer form and retrieves a matching put-form with remove. As a side-effect, the arguments and the code of the selected operation are bound to the local environment of the agent. serve returns a unique form, as the corresponding `result` has to use the same unique identifier. result, finally, puts a unique result-form into the service-space with add.

5.9 Bibliographic remarks

LINDA has longly suffered from the lack of a formal definition of its semantics. Implementations had – for example – to choose a specific semantics for `eval`. Given an operation `eval(f(a), g(b))`, it was unclear, if the active fields were to be evaluated

Laura(host) = Laura-forms(host) + bag-machine(UForm) +
operations
service: Form → Form
service-put: Form × Uid → UForm
service-get: UForm → Form
serve: Form → UForm
result: UForm → Form
equations
f ∈ Form, u, v ∈ UForm
service(f) = service-get(service-put(unique(f)))
service-put(u) = add(u)
service-get(u) = v, v=strip(remove(u)), iff resbound(strip(v),u)
serve(f) = v, v=remove(unique(f)), iff argsbound(strip(v),u) ∧
fieldbound(opcode(strip(v)),opcode(strip(u)))
result(u) = strip(add(u))

Figure 5.18: LAURA's operations

concurrently or sequentially from left to right. Some implementations restrict `eval` to contain only one expression and constants in the other fields. Furthermore, the question of the environment in which these evaluations should take place, was left open. Both procedures in the example could have side effect – do they share variables?

The effects that can be caused by this unclear semantics cumulate with another omission: What sort of expressions can be used in active tuple-fields? If tuple-space operations can be executed during their evaluation and if in our example, `f(a)` contains a sequence ...`out(a);in(?b:int)`... and `g(b)` contains ...`in(?a:int);out(b);b=b+1`..., then the evaluation either locks for any sequential evaluation of the fields or leads to a non-deterministic result for `b` if the environment is shared under a concurrent evaluation.

After the informal definitions in the referenced papers and in [Narem Jr., 89], [Jagannathan, 90], [Jagannathan, 91] gave a first formal definition by the Yale-group. The papers define an operational semantics by a set of rules for term-rewriting. The work also includes semantics for multiple tuple-space and a formal definition for viewing tuples being ordered under a subtype relation.

A thorough investigation in formal aspects of LINDA-specification can be found in [Ciancarini and Jensen et al, 92]. Operational semantics are given as an SOS by transition rules, by a translation of LINDA into CCS without and with value-passing, Petri-nets and as a Chemical Abstract Machine (see below). A more abstract semantics is given as a testing equivalence.

With the exception of Petri-nets and the CHAM, the specifications are based on an interleaving of transitions. For Petri-nets the authors discuss the problems of combining larger nets from small ones and the lack of expressibility of hierarchies, e.g. those imposed by multiple tuple-spaces. [De Nicola and Pugliese, 93] reports on an observational semantics.

A specification of a LINDA-variant can be found in [Hasselbring, 92], [Hasselbring, 93a] using the set-based specification language Z. As Z knows no processes, an interleaving model is incorporated in the specification. The work is very detailed and defines for example, how waiting tuples are searched for a match when an `out` is performed.

On a similar level of detail, [Jensen and Riksted, 89] give an algebraic specification for LINDA and also discuss the consequences of assuming LINDA-operations as atomic and using an interleaving model of processes.

[Chiba and Kato et al, 91] investigate on weak consistency constraints applicable to tuple-space operations in order to optimize protocols for distributed implementation. They define consistent tuple-space histories in which 1) an `out` has to precede an `in` or `rd` that operates on the generated tuple, 2) an `in` is never followed by an `rd` on the same tuple and 3) two `in`'s never operate on the same tuple. Following their intent, they use these condition for the selection of feasible protocols. They do not, however, formally investigate in the notion of the "same tuple" as we do by "instances".

The issue of abstract data types including non-deterministic operations is discussed, for example, in [Subrahmanyam, 81]. Our Bag-Machine contains such an operation, namely remove. It is nondeterministic as its outcome depends on the availability of elements and on the nondeterministic choice of a matching element to remove.

The referenced work allows nondeterministic operations in an abstract data type by defining a characteristic predicate on the operation. The predicate, given an argument list and a value tells if the operation can evaluate to the given value. Using this deterministic predicate, Subrahmanyam develops a notion of equivalence of instances of a type containing nondeterministic operations. For our nondeterministic Bag-Machine, the characteristic predicate can easily be derived from the matching relation: P_{remove}(e,f) = $((e, f) \in \text{match})$ for all elements e and f. [Matthews, 88] adopts the work of Subrahmanyam and discusses the translation of a nondeterministic data type to a CCS specification.

Event structures have been used to give true concurrent semantics to TCSP in [Loogen and Goltz, 91] and [Baier and Majster-Cederbaum, 94] involving operators on event structures similar to ours.

ϵ-structures were introduced in [Mahr and Sträter et al, 90] together with a first order ϵ-logic to build ϵ-formula with which one can speak about objects in ϵ-structures. [Sträter, 92] introduces a logic on ϵ-structures with a total truth predicate and a detailed investigation in self-reference. [Pooyan, 92] uses ϵ-structures to construct semantic models for the λ-calculus. [Mahr, 94] proposes a uniform framework for disciplines of declarations and types. It distinguishes a denotational view in which ϵ-structures are introduced as semantic models and a constructive view using judgements and rules.

Chapter 6

An experimental implementation of Laura

To demonstrate our approach in practice and to prove implementability, we implemented an experimental Laura system. In this chapter we outline the structure of our implementation.

This Laura implementation allows it to perform experiments with our approach in a UNIX-environment. It uses a communication infrastructure provided by the ISIS-toolkit ([Birman and Schiper et al, 91], [Birman, 91], [Birman and Cooper et al, 90]). Two embeddings are provided, one for the programming language C and one for the script language of `csh`. A precompiler exists for these embeddings. External type systems are provided by ISIS and by the external data representation XDR by Sun ([SUNa], [SUNb]).

Being an experimental prototype, the implementation has several restrictions compared to what we defined in the previous chapter. Some of these are dictated by the accessibility of hardware, some by available time-resources and some restrictions stem from the focus of our work.
Our prototype consists of the following components:

- A precompiler that translates programs containing embedded Laura-operations and -definitions into valid programs of the host languages. The precompiler translates programs written in C-Laura and scripts in `csh`-Laura.

- A library that provides functions for the local communications of agents with the bag-machine. For `csh`-Laura the library functions are provided by separate programs.

- An implementation of the Bag-Machine that is instantiated with LAURA's forms and LAURA's matching function.
- An infrastructure for the distributed communication amongst Bag-Machines.
- Some utility programs for analysis and monitoring.

In the following we describe the components as they are implemented for the experimental prototype.

6.1 A C-Embedding: C-LAURA

We chose the language C as a host language for an embedding of LAURA. C is a typical representative for a strongly typed imperative programming language. Its wide-spread use makes it an attractive candidate. The concrete syntax of the embedding is defined in figure 6.1[1]. We will refer to this embedding as "C-LAURA" in the following.

Service-Type ::= `/*SERVICETYPE` Name Service-Type-Declaration `SERVICETYPE*/`

Service-Call ::= `/*SERVICE` Name `(` Operation-Binding `). SERVICE*/`

Serve-Call ::= `/*SERVE` Name Opcode-Var `(` Operation-Bindings `). SERVE*/`

Result-Call ::= `/*RESULT` Name Opcode-Var `(` Operation-Bindings `). RESULT*/`

Operation-Bindings ::= Operation-Binding { `;` Operation-Binding }

Operation-Binding ::= Operation-Name `:` Bindings `->` Bindings

Bindings ::= Variable-Name | Bindings { `*` Bindings } |
`<` Bindings { `,` Bindings } `>` | `[` Bindings { `,` Bindings } `]`

Figure 6.1: Concrete syntax of the C-LAURA embedding

The syntax of the embedding uses C comments to embed LAURA's operations. Thus we are free to use our own syntax and do not have to align it to C-syntax. An example for a C-LAURA-program text can be found in figure 6.4. Moreover, C-LAURA-programs can be compiled for testing purposes by a normal compiler as the coordination-operations then are ignored.

The binding lists define from which program variables the argument values are to be read and to which variables result values should be bound. The mappings from LAURA's type system to and from that of C are defined as follows:

[1] In the concrete syntax we use a `typewriter`-font for terminals.

LAURA		C
boolean	⇔	`int`
number	⇔	`double`
number	⇐	`float`
number	⇐	`long`
number	⇐	`int`
character	⇔	`char`
string	⇔	`char[]`

Two things are worth mentioning. First, C does not have a type boolean, however it is usually implemented as `int`. Whenever a **boolean** can be expected from the service interface type, we map an integer. Second, all numerical types are mapped to a **number**, but **number** is mapped to `double` only. In a second stage of mapping, the `double` then is mapped to the type of the numerical variable as it is assigned. This may lead to range-violations and is not checked by the prototypical embedding. Formally, C-LAURA is an implementation of the embedding Laura(C,XDR).

6.2 A csh-embedding: `csh`-LAURA

As a second embedding, we implemented an embedding called `csh`-LAURA for use with the UNIX-shell `csh`. With `csh`-LAURA we demonstrate the following aspects:

- **Use of multiple programming languages.** C as a compiled programming language and `csh` as a script-language are host-languages for LAURA that have a very different purposes. While C is a universal programming language, `csh`-scripts are rather control oriented to automize jobs on a high level. The typing systems differ radically, as `csh`-variables are untyped. Moreover, whereas C is usually compiled, `csh`-scripts are interpreted.

- **Integration of existing components.** The intention of writing a `csh`-script mainly is to automize the invocation of programs. Thus, with `csh`-LAURA it is easy to integrate applications in an open environment. The script provides a "capsule" that on one side performs coordination by LAURA-operations and on the other side computation by calling programs with appropriate handling of arguments and results.

The syntax of the `csh`-LAURA embedding is similar to that of C-LAURA – it is shown in figure 6.2. In contrast to C-LAURA, scripts cannot be executed without preprocessing, which is caused by the `csh`-syntax that does not know grouped comments. To illustrate the usage of a service with `csh`-LAURA, figure 6.3 shows the usage of the booking-service.

The necessary mappings to LAURA's type system cannot be made specific for the `csh` embedding, as the host language has no type system in this case. Thus any variable can be used for any value of any of LAURA's types and vice versa. Formally, `csh`-LAURA is an implementation of the embedding Laura(`csh`,XDR).

Service-Type ::= `#SERVICETYPE#` Name Service-Type-Declaration `#SERVICETYPE#`

Service-Call ::= `#SERVICE#` Name `(` Operation-Binding `).` `#SERVICE#`

Serve-Call ::= `#SERVE#` Name Opcode-Var `(` Operation-Bindings `).` `#SERVE#`

Result-Call ::= `#RESULT#` Name Opcode-Var `(` Operation-Bindings `).` `#RESULT#`

Operation-Bindings ::= Operation-Binding { `;` Operation-Binding }

Operation-Binding ::= Operation-Name `:` Bindings `->` Bindings

Bindings ::= Variable-Name | Bindings { `*` Bindings } |
`<` Bindings { `,` Bindings } `>` | `[` Bindings { `,` Bindings } `]`

Figure 6.2: Concrete syntax of the `csh`-LAURA embedding

LAURA		`csh`
boolean	$\Leftrightarrow$	*variable*
number	$\Leftrightarrow$	*variable*
character	$\Leftrightarrow$	*variable*
string	$\Leftrightarrow$	*variable*

6.3 The STL-precompiler

For the embeddings we use a precompiler approach. The precompiler `stl` takes a source file with the embedded LAURA operations and generates a translated program that can be compiled by the C-compiler or executed by `csh`. The precompiler has to perform four major translations:

1. For a **SERVICETYPE** definition:
   - Flatten the type definition by expanding the types defined in the **where**-part. By doing so, all names in argument- and result-lists are forgotten. The purpose of the **where**-part is just to introduce names for convenience. They are not used by LAURA's type system.
   - Generate a type definition in an internal representation used by the matching-routine and emit appropriate definitions for the interface types and the operation numbers.
2. For a **SERVICE** operation:
   - Emit statements that copy values from variables according to the binding list into an argument list for a form.

```
# Example agent that uses a travel-agency service

#SERVICETYPE# travel_b
(getflightticket: ccnumber * date * dest -> ack * price;
 getbusticket : ccnumber * date * dest -> ack * price * line)
where
ccnumber = string;
date = <day,month,year>;
day = number;
month = number;
year = number;
dest = string;
ack = boolean;
line = string;
price = <number,number>.
#SERVICETYPE#

set cc = "123456"
set theday = 10
set themonth = 3
set theyear = 1994
set dest = "New York"

#SERVICE# travel_b
(getbusticket : cc * <theday,themonth,theyear> * dest ->
                ack * <dollar,cent> * line;).
#SERVICE#

echo $ack $dollar $cent $line
```

Figure 6.3: Service usage in csh-LAURA

- Emit a service-call to the LAURA-library.
- Emit statements that copy values to variables according to the binding list from a result list from a form.

3. For a **SERVE** operation:

   - Emit a serve-call to the LAURA-library.
   - Emit statements that copy values to variables according to the binding list from a result list from a form.

4. For a **RESULT** operation:

- Emit statements that copy values from variables according to the binding list into an argument list for a form.
- Emit a result-call to the LAURA-library.

For the `csh`-embedding the calls to the library are replaced with calls to stand-alone programs with the argument list as program parameters. An example of the translation is illustrated in figure 6.4 for a C-LAURA-program. The emitted statements are macros that are expanded during the compilation.

The precompiler performs scanning and parsing using a scanner generated by `flex` ([Paxson, 92]) and a parser generated by `bison` ([Donnelly and Stallman, 92]). It does not perform any checking on the types of variables used in the bindings. This prototypical behavior would have to be replaced by a modified C-compiler for C-LAURA.

6.4 The LAURA-library

Any agent in C-LAURA is linked with the LAURA-library; for `csh`-LAURA-scripts three programs offer the library-functionality. For each embedding some form of the LAURA-library has to be provided, as the conversion to and from the external data representation is language dependent. The library performs the following tasks:

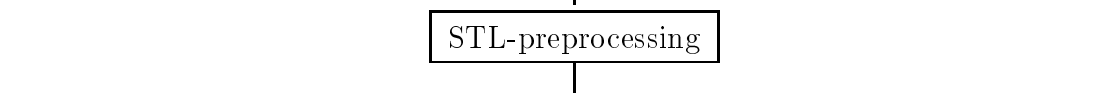
- Construction of forms from argument lists. This includes the proper setup of a data structure, the management of unique identifiers for forms and the construction of command-forms to be communicated to the Bag-Machine.
- Communication of forms to and from the Bag-Machine. This communication is performed in the prototype with UNIX-sockets. Performing an add-operation means to deliver a command-form via a defined socket offered by the Bag-Machine. A remove involves the installation of a socket via which a matched form is returned and the transfer of its address to the Bag-Machine.
- The conversion of data contained in argument- and result-lists to the external data representation for LAURA's types. This conversion is done by using the external data representation XDR ([SUNb], [SUNa]) and the associated operations. This choice also defines the implementation of LAURA's type system for agents working on one machine. It is necessary to introduce an external representation on one machine as agents can be programmed in different programming languages.

Figure 6.5 illustrates how a set of agents running on one machine communicate with the Bag-Machine.

6.5 The Bag-Machine instance

In 5.2 we defined the Bag-Machine and its behavior in a way that abstracts from forms and LAURA's matching. The experimental prototype uses an instantiation of the Bag-Machine and provides the definition of forms and a function that performs the matching and instantiation of forms.

```
...
/*SERVICETYPE travel_b
(getflightticket:  ccnumber * date * dest -> ack * price;
getbusticket:  ccnumber * date * dest -> ack * price * line)
where
ccnumber= string;
date= <day,month,year>;
day= number;
month= number;
year= number;
dest= string;
ack= boolean;
line= string;
price= <number,number>.
SERVICETYPE*/
...
/*SERVICE travel_b
(getbusticket:  cc*<thedate.day,thedate.month,thedate.year>*dest -> ack*<dollar,cent>*line;).
SERVICE*/
...
```




```
#include "laura.h"
...
#define getflightticket 0
#define getbusticket 1
#define ST_travel_b "getflightticketS<NNN>S-B<NN>getbusticketS<NNN>S-B<NN>S"
...
PREPARE_AR;
START_BUILD_ARGS;
MAP_TO_ARGS((cc));
MAP_TO_ARGN((thedate.day));
MAP_TO_ARGN((thedate.month));
MAP_TO_ARGN((thedate.year));
MAP_TO_ARGS((dest));
END_BUILD_ARGS;
CALL_SERVICE(ST_travel_b,getbusticket);
START_READ_RES;
MAP_FROM_RESB((ack));
MAP_FROM_RESN((dollar));
MAP_FROM_RESN((cent));
MAP_FROM_RESS((line));
END_READ_RES;
FINISH_AR;
...
```

Figure 6.4: A service-request in C-LAURA pre-processed by the STL-compiler

Forms as handled by the Bag-Machine consist of a form-type, a unique identifier, a service-type encoding, an operation selector and a list of arguments or results. The form-type indicates whether the form is a request for a service (`service-put`), a request for the service-result (`service-get`), a service offer (`serve`) or a service result (`result`). The unique-identifier makes forms unique so that a result can be directed

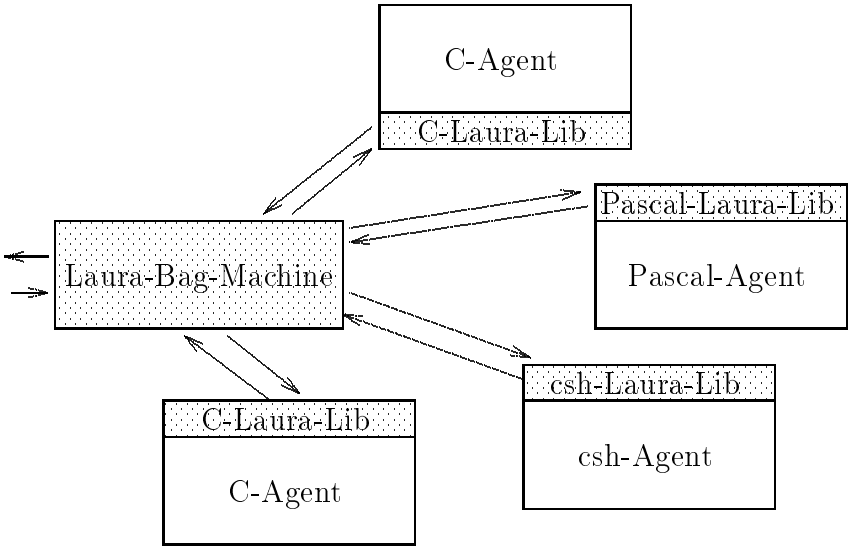


Figure 6.5: Agents communicating locally on one machine

to the agent that requested a service. Unique identifiers can be easily generated from the process number of the agent and a counter held in the library.

The service-type encoding is generated by the precompiler and can be matched faster than the representation in the STL-grammar. Argument- and result-lists consist of pairs of a code for a LAURA's type and a value in the external data representation.

The matching takes advantage of the form-types. A test for the subtype-relation on service-type encodings is performed only when matching `service-put-` and `serve`-forms. For the case of `service-get-` and `result`-forms only the unique identifiers are tested for equivalence. All other combinations of form-types match in no case.

The test for subtype-relation on service-types works on the encoding of the service-type according to the subtyping-rules defined in section 5.1.2. As the order of fields can be rearranged, this function also keeps track of necessary re-orderings in argument- or result-lists in form of a "copylist". When instantiating the values of a matched form, the copylist is taken into account.

The Bag-Machine's implementations of add and remove are independent of the out-form of elements and the matching-function. For our prototype, the Bag-Machine uses two pools of elements, the add-pool and the remove-pool. Elements that "wait" for a matching element are stored in the remove-pool together with information where the matching element should be delivered. Elements not yet removed are stored in the add-pool. The pools are organized as hashed lists that are accessible via some identifier. Elements in the add-pool are hashed by some unique identifier, those of the remove-pool by the information about the agent a matching element should be returned to.

When an add is performed, the elements in the remove-pool are searched sequentially, if the new element matches some "waiting" element. If so, the added element is delivered, thus completing a remove. If no match is found, the element is entered to the add-pool.

When performing a remove, the add-pool is searched sequentially for a matching element. If none is found, the element passed to remove is entered to the remove-pool. If the Bag-Machine is distributed, distributed searches for matching elements are performed prior to the insertion to the pools.

The approach we use has two important characteristics. First, the selection of matching elements is fair as the "oldest" elements are searched first. Second, a local search is performed prior to a distributed search.

The Bag-Machine-implementation can be optimized in two ways. First, a function can be defined for elements that return a characteristic part of the element. This then is used for a hashing function that points to a subpool of elements. By introducing multiple add- and remove-lists, the length of the searches would be shortened. Second, the Bag-Machine should take advantage of the independencies defined in 5.2 by using light-weight threads that perform add- and remove-operations concurrently. For the experimental implementation this is partially done as local and distributed operations are performed using a threads-package.

6.6 A distributed Bag-Machine

The Bag-Machine implements operations that work on a shared collection of element. For the case of a centralized implementation on one machine, two pools of elements are held and operations on them suffice for the implementation.

This scheme could be extended to a naïve distributed implementation with two sorts of Bag-Machines: One central Bag-Machine on a single machine that holds the complete pools of elements and a number of proxy-Bag-Machines that simply forward add- and remove-operations to the central Bag-Machine. Such a scheme would lead to a bottleneck and would introduce a sequentialization of all distributed accesses.

Two other implementation schemes could be chosen: One in which all added elements are replicated on all machines and one in which no elements are replicated. For the first scheme, the costs for maintaining consistency rise with the number of nodes, as the replicas have also to be removed consistently. The second scheme involves a communication overhead, as any distributed handling of elements – i.e. when a matching element is not found in the local search – involves a request for all other nodes to start a search and a positive or negative answer.

In order to find a compromise between these cost-extremes, we chose a replication scheme similar to the one originally proposed for the S/NET Linda-implementation. The idea is to replicate elements only partially on subsets of all nodes. Within this subset, the search for an element can be performed locally. In contrast to a full replication, the costs for removing a replica decrease. In order to find an element from all subsets, only one node per subset has to be asked for a matching element. This decreases the communication costs.

The distribution scheme is as follows. Let N be the set of nodes that participate in the system. Subsets of nodes can form logical busses meaning that they have the availability to send and receive broadcasted messages to and from the nodes of this subset. There is a set of logical busses $A = (A_0, \ldots, A_n)$, called *add-busses* and a set of

logical busses $R = (R_0, \ldots, R_n)$, called *remove-busses*. A node from N is a member of exactly one add- and one remove-bus, so that $\bigcap_{i=0}^{k} A_i = \emptyset$, $\bigcap_{i=0}^{k} R_i = \emptyset$, $\bigcup_{i=0}^{k} A_i = N$ and $\bigcup_{i=0}^{k} R_i = N$. Add- and remove-busses are organized so that they form a grid in which an add-bus intersects all remove-busses and vice versa.

Given such an organization, the compromised replication scheme can be implemented by replicating an element that is added at some node over the add-bus which the node is part of and by trying to remove an element from the nodes of the remove-bus. As the remove-bus intersects all add-busses, the union of the replicas held by the nodes therein equals to the union of all elements that have been added in the system. We use this approach in our experimental prototype.

Moreover the organization has the advantage to localize the effect of an add – all add operations on distinct add-busses are completely independent. Also, searches on distinct remove-busses require no synchronization prior to the removal of a replica. Doing so takes advantage of the independencies formally defined in the previous chapter.

A request for a remove on a remove-bus does access all elements that are currently in the system. If the request fails, the remove-operation has to be stored in some remove-pool from which it is periodically re-issued on the remove-bus. This approach can be seen as a polling- or active-wait-operation, which is generally considered an unelegant technique in distributed programming. But, as we are dealing with open distributed system, in which a dynamic structure is a given fact, such re-issues are necessary in any case, since they have to be performed at least for new, joining nodes. Moreover, the delay between an add of an element and its removal in the next re-issuing-cycle can be accepted under the relaxed efficiency requirements given.

The organization is depicted in figure 6.6 for an example system. The nodes are represented by large circles with an element-storage contained in them. The lines represent logical busses as just described. Elements A, B, C, D and E exist in the system consisting of nine nodes. There are three add- and three remove-busses. Elements A and B have been added on nodes of add-bus 0 and are replicated on all the nodes connected to the bus. C was added by some node of add-bus 1 and D and E on add-bus 2. The different layout of the element-stores reflects the fact that no distributed shared memory is established, as elements are not referenced by addresses. The local organization of this storage can be different from node to node.

Now, any node has access to all elements in the system by requests on the remove-busses, as the union of the element-stores on all nodes of a remove-bus is A, B, C, D and E in all three cases. The organization also allows for a parallel removal of as many distinct elements as there are remove-busses, even with Bag-Machines that operate purely sequentially. The three nodes in add-bus 0 can remove elements A, C and E for example without any interference. Also, as many replications can be performed in parallel without any threading within the bag-machines as there are add-busses.

However this organization always requires $n \cdot m$ nodes for n add- and m remove-busses, thus degenerating the system to one add- or remove-bus for – say – 7 nodes which, in turn, corresponds to the organizations with full or no replication as described above. When distributing a conventional program, the number of nodes involved can be chosen in advance to the execution. For an open system, however, it changes dynamically at runtime.

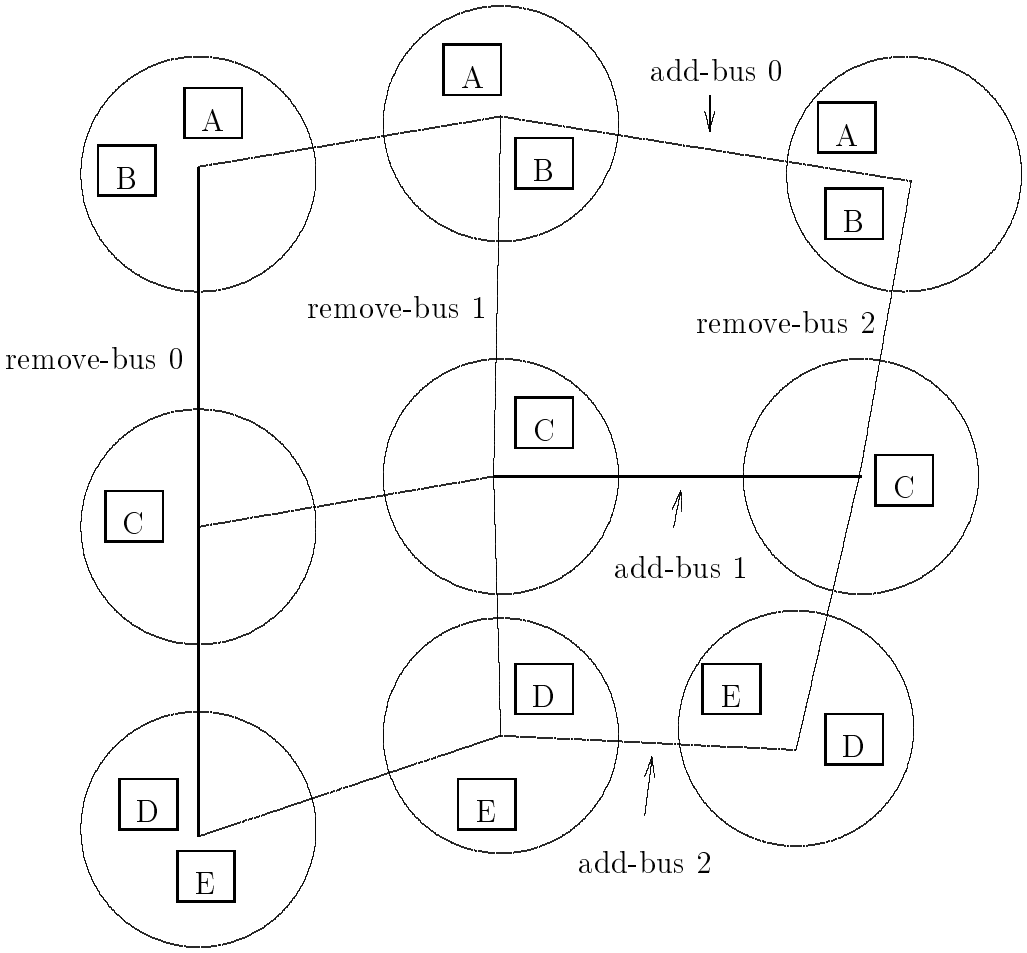


Figure 6.6: A partial replication scheme

We therefore introduce the notion of *pseudo-node* and design a protocol that handles joining and leaving nodes in order to maintain the described organization while meeting the requirements of open distributed systems.

As described, a node is member of exactly one add- and one remove-bus. This requirement can be upheld for any number of nodes, if we allow a node to have multiple identities in the system. We speak of a *pseudo-node* for a member of a bus that is simulated by another member on the same add-bus. This simulation is easy to achieve, as the element-replication on the add-bus requires no overhead and the simulating node only has to join a second remove-bus and handle remove-operations from there, too.

Above we stated that we use the ISIS-toolkit as a communication-infrastructure for our experimental prototype. ISIS is a toolkit that uses *process groups* and *broadcasts* as the basic mechanisms to implement communication and synchronization in a distributed environment. A process group is a collection of processes that can be located on any node in the ISIS-system. A process can join and leave symbolic named groups. Communication is performed by broadcasts of messages to groups. These messages then are delivered to all members of the group.

Some characteristics of ISIS are:

- ISIS offers various broadcast primitives that follow different consistency constraints. It can be adjusted, if the delivery of a message is totally ordered with respect to all broadcasts or to those addressed to a specific group. Also, broadcasts can be delivered in a lazy fashion when their order is uncritical.
- ISIS is fault-tolerant in that it implements a number of logging-mechanisms to ensure broadcast-consistencies in the light of errors.
- ISIS offers a machine-independent data-representation, thus supporting heterogeneous process groups.
- A set of basic synchronization primitives is implemented within the ISIS-toolkit, such as support for locking. Also, some utility-functionalities – such as an automated state-transfer to agents joining a group – are offered.
- A package for non-preemptive lightweight-threads is provided that allows it to make the handling of received messages and other activities multi-threaded.

The distributed organization we outlined in this section, fits well on ISIS mechanisms. We can map the logical busses directly on process-groups. Broadcasts then are messages sent to a process-group. The fault-tolerance mechanisms ensure that no broadcasts will be lost, which would be disastrous for our organization. The synchronization mechanisms ease greatly the implementation of the protocols described in the next sections.

In this thesis we are concerned with the coordination of services in open distributed systems, not with the implementation of a communication infrastructure. But the choice of a package that provides us with a complete communication infrastructure does not mean to put aside an important part of such systems. In contrast, this is a natural approach as the integration of existing components is one of the main goals with open systems.

At the time of writing, the major computer- and operating-system manufacturers undertake numerous activities to introduce such a communication infrastructure at a logical higher level as standard components in future workstation operating systems. An example for a consortiated effort is the Object Management Group's (OMG) Object Management Architecture, which defines a reference architecture for an object-bases infrastructure ([Soley, 93], [OMG91]). Amongst the implementations of this OMG model is IBM's System Object Model ([IBM93]).

Other approaches are, for example, Sun's Distributed Objects Everywhere (DOE), which will be based on the technology of the operating system NeXT-Step and be marketed as OpenStep ([NeX94]).

We can extrapolate these industry-developments to a scenario in which all modern computing equipment comes with a communication platform on a high logical level. An open system should make use of these platforms as is should do today with existing TCP/IP platforms. With these, our choice of the communication infrastructure ISIS simply reflects an assumption that is valid for the next versions of major operating systems.

The distributed organization of the Bag-Machine is dynamic in order to allow joining and leaving agents at any time. When an agent requests a Bag-Machine on a machine which has no Bag-Machine running, it is started automatically by a library. The local Bag-Machine starts up and joins the distributed organization by a protocol which is described in section 6.6.2. The next subsection describes the protocols used in the distributed Bag-Machine for add and remove.

6.6.1 Protocols used for the distributed Bag-Machine

Three protocols are necessary for the distributed execution of add and remove with the grid-architecture: the addition and the removal of a replica on the add-bus and the removal of an element from the remove-bus.

Most simple of them is the protocol for add – it requires one broadcast of the new element on the add-bus. The broadcasted message is `add-replicate(e,r)`, where `e` is the element to be replicated and `r` an identifier unique on the add-bus. The identifier can be easily generated from the nodes network address and a counter held in the Bag-Machine.

In consequence of the receipt of a replica, all nodes search their local remove-pool for a match. If one is found and a remove of the replica on the add-bus with the protocol below succeeds, the element is returned to a waiting agent.

Removing a replica involves a two phase protocol. First, a request for a lock of the replica is broadcasted on the add-bus as `lock-replicate(r)`, where `r` is the unique identifier of the element. If the element is not currently offered for a remove on an remove-bus by the protocol below, a flag is set in all replicas preventing the element from removal. If the lock succeeds, a `remove-replicate(r)`-message is broadcasted causing all nodes on the add-bus to delete the element identified by `r` from the add-pool.

For a remove-operation, the local add-pool is sought for a matching element. If one is found, an attempt is made to remove the replicas on the add-bus with the protocol just described. If none is found or if all removals on the add-bus failed, the nodes on the remove-bus are asked for a matching element. The protocol for the removal of an element from the remove-bus is described below, where the left column contains the operations of the node requesting the remove and the right one those of the other nodes on the remove-bus.

Node requesting the remove	Other nodes on the remove-bus
Broadcast a message `want(e,id)` on the remove-bus. `e` is the element for which a match is sought and `id` a unique identifier of the request. It is generated from the machine-address and a counter.	
	On receipt of a `want(e,id)`-message, search the local add-pool for an element matching `e`. If one is found, lock it on the add-bus with `lock-replicate(r)`-broadcast, where `r` is the unique identifier of the found element on the

addbus. If the lock succeeds, reply with an `offer(r)`-message and memorize the offer in a pool. If no matching element was found, reply with a `dont-have`-message.

Collect the offers and chose one. Broadcast a `send(r,id)`-message, where `r` is the identifier of the chosen offer and `id` the request identifier. If no offer was made, store the element in the remove-pool.

On receipt of a `send(r,id)`-message, check the previous offer to the request `id`. If the request is for the offer, answer with it and broadcast a `remove-replicate(r)`-message on the add-bus. If no element was offered, ignore the message. Otherwise, broadcast an `unlock-replicate(r)`-message for the offered element on the add-bus.

Deliver the received matching element to the agent that performed the remove.

For a system with n remove- and m add-busses, add requires the delivery of n copies of the element and can lead to n requests for a lock in the worst case and n remove-messages. The lock-requests, however, can be partially avoided by local tests.

remove leads to a lock-request and n remove-messages in the best case where a local match is found. In the worst case, however, m copies of the element have to be delivered with the `want`-messages. These can lead to $n * m$ lock-requests and n remove-messages.

The asymmetric loads for add and remove can be justified by the nature of the operations. add is non-blocking and involves nearly no costs. remove is blocking anyway – for the case of LAURA, the blocking lasts at least for the execution of the service-request – so that a coordination-overhead is acceptable.

Making the reasonable assumption that for LAURA a matching form cannot be found on every remove-bus, the worst case should not occur. This short analysis shown however that the structuring of remove- and add-busses should be guided by some additional information. The more "similar" elements are replicated on the same add-bus, the lower the overhead for the remove-protocol on the remove-bus will be.

The toolkit we chose for our experimental implementation, ISIS, provides several variants of broadcasts which are associated different costs. Choosing these variants means to determine the ordering requirements for the delivery of the broadcast messages. The following table shows them.

Message	Ordering required
`add-replicate(e,r)`	Strict ordering as `lock-replicate` requires consistency of replicas.
`lock-replicate(r)`	Strict ordering as a consistency of locks is required.
`remove-replicate(r)`	Relaxed ordering as replica `r` has been locked in advance and is subject to no other operation.
`unlock-replicate(r)`	Relaxed ordering as replica `r` has been locked in advance and is subject to no other operation.
`want(e,id)`	Relaxed ordering as the recipients search their local element-pools which requires no form of consistence and as there are no fairness-guarantees made. Consistent locking on an add-bus in the case of an `offer`-answer is ensured by the strict ordering of `lock-replicate`-messages.
`send(r,id)`	Relaxed ordering as `r` as well as other offers have been locked on add-busses. However, the message should be delivered fast not to hold unnecessary locks too long.

A strict ordering here means that all recipients receive broadcasted messages in exactly the same order. A relaxed order means that the order of reception does not have to be the same for all recipients. The latter allows the communication infrastructure to optimize message-delivery.

6.6.2 Protocols for joining and leaving nodes

The protocols necessary for joining and leaving nodes ensure the grid-structure which is required for the above protocols.

An agent started on a machine causes the automatic startup of a local Bag-Machine which joins the distributed Bag-Machine. It does so by contacting a subset of the nodes that is known as the *reception*. The reception is formed by one node from each add-bus. A node being a "receptionist" has some additional knowledge about the current organization of add- and remove-busses. Moreover, the receptionist node of an add-bus is the only node that simulates other nodes. The protocol for a join works as follows:

Node joining	Nodes from the reception
Initialize the communication infrastructure and join the set of nodes.	
Broadcast a `checkin`-message to the reception.	
	On the receipt of a `checkin`-message, one member of the reception – the receptionist – acquires a reception-lock, so that only one join is being served at a time.
	The receptionist broadcasts a `offer-location`-message to the reception.

On the receipt of a `offer-location`-message, a reception-node checks if it simulates one or more nodes. If this is the case, it answers with a `offering(a,r,w)`-message, where `a` is the number of its add-bus, `r` the number of the remove-bus the simulated node joined and `w` a weight taken from the number of simulated nodes. Otherwise reply with a `no-offer`-message.

There are three possible reactions of the receptionist:

1. If the receptionist receives `offering`-messages, it selects one based on the received weights. It answers the joining node with a `join(a,r,n)`-message, where `a` is the number of the selected add-bus, `r` the number of the selected remove-bus and `n` the address of the reception node of the add-bus `a`.

2. If the receptionist receives only `no-offer`-replies, it can decide to build a new remove-bus. In this case it broadcasts an `enlarge(n)`-message to the reception.

   On the receipt of an `enlarge(n)`-message the nodes from the reception add a simulated node to a new remove-bus with the number `n`.

   After the acknowledge of the `enlarge`-message, the receptionist answers to the joining node with a `join(a,r,n)`-message, where `a` is the number of the receptionists add-bus, `r` the number of the new remove-bus and `n` the address of the receptionist.

3. If the receptionist receives only `no-offer`-replies, it can decide to build a new add-bus. It answers the joining node with a `join(r,a,0)` where `r` is the number of remove-busses that have to be joined, `a` the number of the new add-bus and 0 a null-address.

On the receipt of a `join(r,a,n)`-reply two cases arise:

1. If `n` is a valid address, join the add-bus `a`. During this join, the state of one other node of the bus is transferred. This transfer includes all elements that have been added on the add-bus together with unique identifiers and locking status. Then, join the remove-bus `r`. Here, no state has to be transferred. Finally, tell node `n` that it can refrain from simulating the node (a,r).

2. If `n` is a null-address, a new add-bus numbered `a` is created. There is no state to transfer, as no elements have been added yet. The node then joins remove-bus 0 and simulates $\mathtt{r}-1$ pseudo-nodes that join the remaining remove-busses. Finally, the node joins the reception.

Now the node has joined the grid-organization and signals this by broadcasting a `checkin-finish`-message to the reception, causing the receptionist to release the reception-lock.

Figure 6.7 shows the evolving structure of the grid as two nodes join. Here, solid bullets represent real nodes, while hollow bullets stand for simulated nodes. The reception consists of the members of the remove-bus 0.

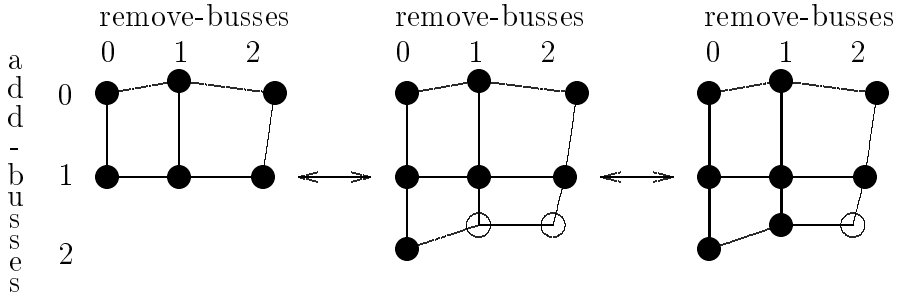


Figure 6.7: The organization evolving by added nodes

In the protocol above the receptionist makes a choice on whether to establish a new remove-bus or to enlarge the system by a new add-bus. For our experimental implementation, we try to hold the number of add- and remove-busses in balance, thus deciding for a new add-bus if there are more remove-busses and vice versa.

For a real system, this decision and the choice of an offered location would be guided by some additional run-time information. If – for example – an add-bus is formed by a number of nodes located closely and replications occur often, is would not be a good decision to direct a node at a distant local to join that add-bus.

Agents can leave the system without further restriction due to the uncoupled style of coordination in the Bag-Machine. Some circumstances can make it necessary or desirable that a local Bag-Machine is shut down when no agents are active. It does so by leaving the distributed Bag-Machine. As for the join, protocols are necessary that uphold the grid-structure of distribution. The protocol for a leaving node is as follows.

Node leaving	Reception node on the add-bus
When no elements are offered on the remove-bus, a check is made if the node is member of the reception. This leads to two cases.	
1. If not member of the reception, leave the remove-bus. Then, broadcast a `checkout(r)`-message on the add-bus, where `r` is the number of the former remove-bus of the leaving node. Leave the add-bus and terminate.	

	On receipt of a `checkout(r)`-message, check if member of the reception. If so, simulate an additional node that joins remove-bus `r`. There is not state to be transferred, as the node holds a complete replica of the state of the leaving node.
Node leaving	Members of the reception
2. If member of the reception and if the last node on the add-bus, acquire the reception-lock. For each element from the add-pool, select another member of the reception and broadcast a `checkout-move(a,e)`-message, where `a` is the address of the selected node and `e` the element of the add-pool.	
	On receipt of a `checkout-move(a,e)`-message, check if `a` is the nodes address. If so, replicate element `e` on the add-bus.
Destroy the add-bus. Leave the remove-bus. Broadcast a `checkout-finish(a)`-message to the reception, where `a` is the number of the add-bus. Release the reception-lock, leave the reception and terminate.	
	On receipt of a `checkout-finish(a)`-message, mark add-bus `a` as unused.

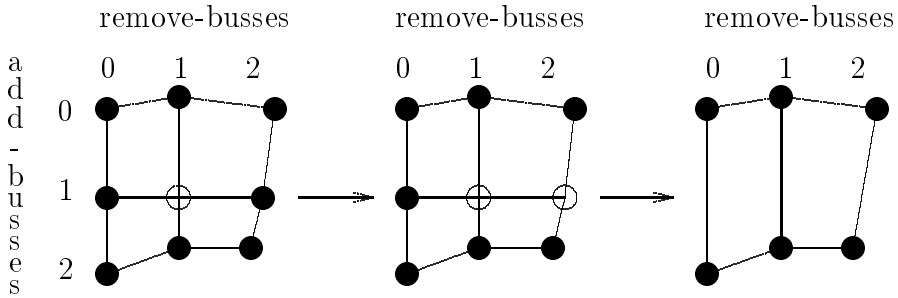


Figure 6.8: The organization evolving by deleting nodes

Figure 6.8 shows the evolution of a grid when two nodes leave. The removal of a node from the system should be subject to some administrative decision. If it is known that no agents will be executed on a machine it can save communication costs to do so, as remaining elements then are moved to a node which is more likely to remove them. Also, a physical shutdown of a node requires the logical shutdown of the local Bag-Machine.

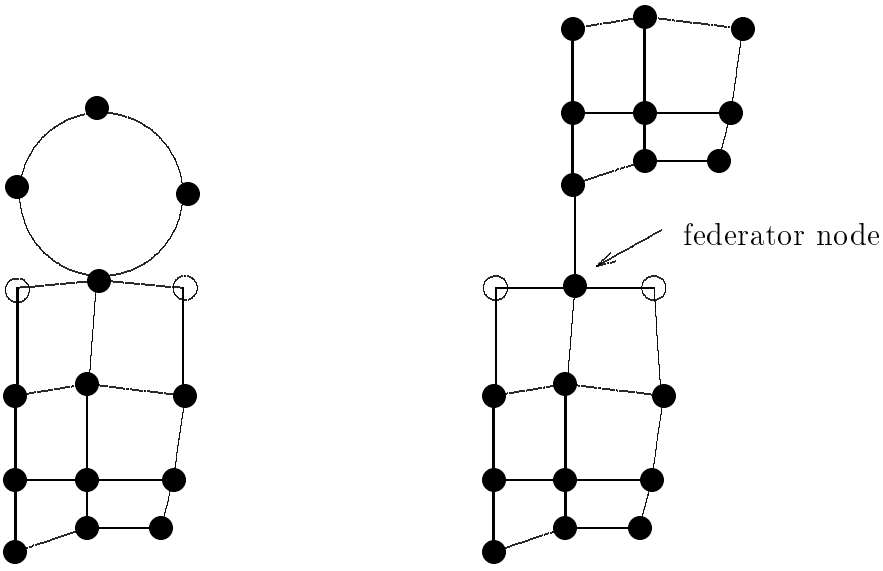
The leave of a node has to be delayed, if elements are currently offered on the remove-bus or if the remove-pool is not empty. In the later case, however, some agents still exist on that node that perform a remove.

In the protocol above, a leaving node that is the last node of an add-bus has to decide on the destination of the `checkout-move`-operation. This decision can be based on additional information on communication-costs or the like. For our experimental prototype, we distribute elements to all other nodes of the reception in order to take advantage of parallel replications on the remaining add-busses.

6.6.3 Extended Bag-Machine-organizations

The distribution scheme outlined relies on a very uniform organization that is only relaxed by pseudo-nodes. Such a uniform organization cannot be guaranteed to be realizable in the light of the heterogeneity and scale of open distributed systems.

It shows, however, that the scheme is not as uniform as it may look. The grid-organization has two components, the replication scheme on the add-busses and the access scheme on the remove-busses. The two components are orthogonal to each other. We demonstrate this by two extended organizations.



(a) Integrating different topologies

(b) Federation

Figure 6.9: Extended organizations

Figure 6.9(a) shows an extended organization that breaks the uniformity of the add-busses. It integrates a logical grid-topology and a logical ring-topology by making one node of the ring a member of all remove-busses of the grid-topology. This is easily achieved by pseudo-nodes.

A requirement of the grid-organization is that a node of a remove-bus can access a subset of elements stored in the system. For an add-bus topology, this is achieved by replication. However, this is not a required organization. If there is some protocol in the example-ring making all elements available for all ring-nodes, then it suffices to

integrate one ring-node into the grid, as illustrated. The protocols on the remove-busses are left unchanged, only that the ring-node does not search its local storage of replicas but starts some protocol on the ring and makes appropriate offers.

Also, the protocols for joining and leaving nodes remain unchanged, as long as the ring-node is member of the reception and makes no offers for unused locations on the grid and observes the protocol for the addition and removal of remove-busses. The ring, however, can have its own protocols for leaving or joining nodes.

This example shows that different topologies can co-exist for a distributed Bag-Machine within the logical grid-organization. It therefore can cope with the heterogeneities of open distributed systems.

Two other characteristics of open system have to be reflected by a distribution scheme: They can cover large areas and they cross organizational borders. Figure 6.9(b) shows how these can be dealt with for a distributed Bag-Machine by federation.

Here, two distributed Bag-Machines are connected by a *federator*-node. It is member of one Bag-Machine by a simulated add-bus as the ring-node above. Also, it has a connection to one node of another distributed Bag-Machine that can be at a different location and belong to a different organization.

The federator serves no local agents but has its purpose solely in connecting to another Bag-Machine. It is able to answer remove-requests by issuing them on the remove-bus of its Bag-Machine. Also, it is able to forward requests to a node of the other Bag-Machine. As any node of a grid is able to satisfy remove-requests, forwarding a request can access all elements stored in another Bag-Machine.

Such an organization satisfies the requirements we described. First, the connection to a remove Bag-Machine can and should make usage of dedicated wide-area communication lines.

Second, federators can be used as "entry-nodes" for Bag-Machines that exist in an organizational context. Recurring to the example with the traveling system of chapter 4, one can imagine that the flight-reservation system is coordinated by a distributed Bag-Machine within the carriers organization and that the banking system used for the authorization of credit cards with one of bank.

Having defined federators at each distributed Bag-Machine allows it to introduce some form of access-control. It could be defined that the bank accepts inquiries from a travel-agency but none of a flight-carrier.

In this case, the federator node of the bank simply does ignore elements coming from the federator of the carrier but handles elements from the travel-agency. Our distribution scheme does not require additional protocols for the denial of operations, ignoring them suffices.

In both extensions of different topologies and of federation, the additional nodes do not necessarily mean the requirement of an additional machine. Still, these schemes are only a logical structure that can be implemented on nodes that run another Bag-Machine-kernel or an extended kernel. However, they impose additional loads and should be selected carefully by an administrative decision.

6.7 Experience with the prototype

From the development of the experimental prototype we learned that LAURA can be implemented without further problems. Some experiences list as follows:

- All components turned out to be implementable fast and easy. It showed that LAURA induces a good structure of small functional units.
- It turned out that the embeddings were easy to implement. This is mainly caused by the focus on coordination and the small interface to the coordination language.
- The components were implementable without being dependent on a specific infrastructure. It is well possible to use another external type system such as ASN.1 or some other communication toolkit.
- `csh`-LAURA turned out to be enabling for the setup of example service-spaces that use and offer functionalities which could be integrated by simple scripts.

For demonstration purposes we defined some agents that together form an open system for flight-, train- and bustickets as described in chapter 4. The agents use the `csh`-LAURA- and C-LAURA-embeddings and simulate a bank, bus- and flight-carriers, travel-agencies and a user-agent. Figure 6.10 on page 107 shows a screen-shot of this demonstrator.

Here, a service-space has been established on five machines, called `flp`, `flp1`, `alice`, `madeira` and `dorothy`. In the lower two windows agents that offer and use a travel-agency have been executed. The lower left window show the output of the `agency`-agent running on `dorothy`; the lower right window contains the output of the `user`-agent running on `madeira`.

The three smaller and two larger windows in the upper half of the screen contain logging messages generated by the Bag-Machine-implementation. The larger ones are log-windows for the Bag-Machines on `dorothy` and `madeira`. They contain entries on the executed protocols and the messages exchanged as described in this chapter. Also visible are the internal representation of forms used in coordinating the booking-service.

6.8 Bibliographic remarks

The distribution scheme for the Bag-Machine is adopted from the one described and analyzed in [Carriero and Gelernter, 86]. Besides the the software-implementation described in this source, it has been realized in hardware for the Linda-machine ([Krishnaswamy and Ahuja et al, 88], [Ahuja and Carriero et al, 88]).

An analysis of consistency requirements for a distributed implementation of a tuple-space has been performed in [Chiba and Kato et al, 91]. For `in` on replicated tuple-pools, it involves a strict protocol with locking, a nonexclusive protocol without locking but with acknowledgments and a weak protocol consisting of an erase-broadcast only. The selection of the protocols depends on the `in-out`-patterns of agents and is adjusted by the programmer or an optimizing compiler.

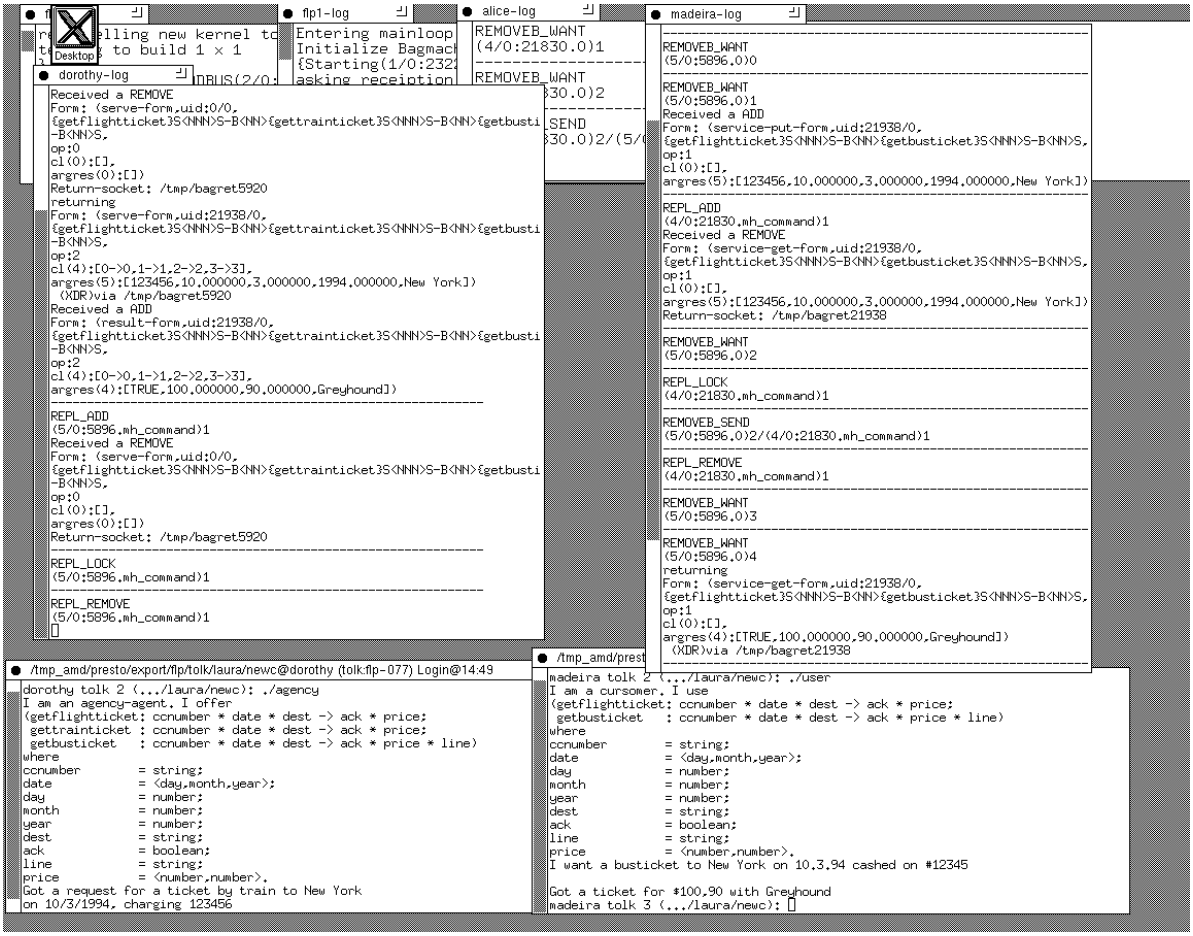


Figure 6.10: Screen-shot of some LAURA-agents

For the Bag-Machine, the remove-protocol on an add-bus is similar to the strict protocol. As we cannot make any assumptions on the behavior of agents, the other protocols provide no alternatives for our implementation.

In [Mattson and Bjornson et al, 92] an alternative for a distributed implementation is described. Here, elements are not replicated but stored on one machine in a network, whose location is determined by a hash-function on the element. Given that elements being in the match-relation have the same result with the hash-function, then only one node has to be asked for a matching element.

For our purposes, this approach is not feasible, as it depends on a static set of nodes. Would new nodes be introduced, then the range of the hash-function would have to be changed. Moreover, the already distributed elements probably have to be moved, so that the join of a node could lead to a complete reorganization of the distributed element storage. The same argument applies when nodes are allowed to leave. Given the potential world wide scale of open distributed systems, such reorganizations will become impractical and should not be part of an architecture.

A Linda-like system based on the ISIS toolkit is reported in [Westbrook and Zuck, 94] together with an analysis on the communication- and replication-costs in the presence of faulty machines. The communication infrastructure P4 ([Butler and Lusk, 92]) is used in [Butler and Leveton et al, 93] to implement a Linda system in two flavors. In a shared memory implementation, agents perform Linda's operations on a shared tuple storage whereas in a memory passing implementation, a single process acts as a tuple-space manager. The PVM-infrastructure ([Beguelin and Dongarra et al, 91]) serves as the implementation platform for a Linda system called Glenda in [Seyfarth and Bickham et al, 94]. All references report on work in progress but support our claim that other infrastructures than ISIS can be used for a Bag-Machine implementation.

Chapter 7

Outlook and perspectives

In this thesis we introduced a model of coordination and discussed the three aspects of coordination, communication, synchronization and service-usage and -provision. We studied and proposed three languages, namely Linda, Alice and Laura, which each put emphasis on one of these aspects. They share the basic concept of the linguistic treatment of coordination separated from computation. Also, they all three use uncoupled coordination by a shared multiset of some elements as the basic mechanism. We have shown that the conceptual separation of coordination as well as the chosen coordination-media are enabling to a solution of the coordination problem in its three aspects.

We have given a formal prescription of coordination with multisets based on the Bag-Machine by specifying its behavior and that of agents using it. A formal specification can be used to verify an implementation and to establish a notion of correctness.

What remains to be performed, is a validation of our approach, i.e. exploring whether the solutions we proposed here are capable of solving coordination problems as given by the reality. Such a validation necessarily includes non-formal methods.

For Alice, we did not touch the issue of implementing the language. Some pretentious requirements are imposed. Most important, a suitable representation of process-definitions has to be found. On one hand, Alice-processes can be compiled for some architecture, involving a mixture of Alice-operations and local computation. However, as process-definitions can be communicated in tuple-fields, there has to be some architecture-independent representation of these definitions which cannot be compiled in advance when multiple architectures are involved. It would have to be evaluated, if a two-folded representation with executable machine-dependent code and interpretable

machine-independent code is suitable for the implementation of a coordination-assembler that imposes higher efficiency requirements.

Also, we only outlined how a simple imperative language can be compiled into ALICE-agents. It remains to work out the details of further mechanisms found in a real language and to find representations of data-structures in the local-environment of an ALICE-agent.

In the case of LAURA, the implementation of a real prototype – instead of an experimental system – has to provide a starting point for a validation of our approach by a case-study. It would have to pay more attention to efficiency and had to be validated against solutions using an ODP-framework or an OMG-like platform. Such a prototype has to take into account a variety of system-parameters which we neglected for our experiments. Examples are factors such as real-life loads on communication infrastructures or the integration of preexisting real components.

Two important issues in open systems are not addressed by LAURA, quality of services and management. Quality of services means that multiple offers of the same service are distinguished by some measure of quality, such as the resolution of a printer or the expected duration of the service provision. In approaches such as ODP, some set of service-attributes is defined and can be used to formulate quality-requirements when requesting a service, such as `DPIResolution>=300`. Selection of a service then means to find appropriatly typed service offers and then to chose one according to the quality requirements. For LAURA, such a mechanism can be added by introducing additional quality-terms in forms and by extending the matching-rule for forms. However, this will introduce the naming problem for attributes again, as `DPIResolution` and `ResoutionInDPI` would have to be related.

A second important issue not addressed by LAURA is the management of the system. Management deals with how to cope with performative problems such as balancing the usage of resources at runtime. For LAURA this means to manage the Bag-Machine by gathering information about – for example – the number of forms stored in the addpool and by providing management function that allow it – for example – to redistribute forms to other addbusses if there is an exceptional amount of forms added. Thereby storage- and communication-loads could be better balanced. Management can also be supported by additional knowledge about what services will be used and offered by an agent. This information can be passed to the Bag-Machine by initially transmitting the servicetype definitions. We addressed some of these issues in the chapter about the experimental prototype but still have to establish some management-concept for LAURA.

Given the developments in information technology, multi-media requirements raise further questions that are not addressed in LAURA and not yet answered in other approaches. An example is how to deal with services that require a stream-like dataflow such as the transmission of video- or audio-data. Also, we addressed how safety issues can be implemented in a LAURA-system, but still a validation with respect to real-life requirements of organizational structures in necessary.

Open distributed systems, which provide the context to the main contribution of this thesis, are a field of research that uses the notion of frameworks to provide abstract models that are able to cope with the enormous complexity induced by the variety of

issues considered important. Examples are – most prominently – the ODP reference model, ISO's quality framework ISO 9000 ([ISO, 91]), or – with a broader pretension – the multi-dimensional Δ-model ([TRI94]).

We focussed on the issue of coordination which is only one item within such frameworks. Our approach to coordination in open distributed systems has to be validated against such frameworks. It remains to be shown how our understanding of coordination has counterparts in these models. The conceptual separation of coordination and computation we used has to be checked against the finer-grained separation of interests in those models, which also has the potential to address the "further issues" we mentioned above with a clearer view on their role in systems.

Besides of a validation of our approach against broader models, we envisage a verification of out model of coordination against other approaches to coordination which chose a different coordination-ontology, a different coordination-media and associated rules. In fact, it remains to show how the ODP- or OMG-approach to coordination in open distributed systems is an instance of our model. With a theoretic perspective, we would have to develop formal means to show relations between our Bag-Machine and formal models of communication and synchronization.

Chapter 8

Acknowledgments

Naturally, the work on thesis has been dependent on an organizational infrastructure. The institutional context has been provided by the Graduiertenkolleg Kommunikationsbasierte Systeme at the Technische Universität Berlin, Freie Universität Berlin and the Humboldt Universität zu Berlin. The author is indebted to the professorial members of the Graduiertenkolleg during the past three years, Hartmut Ehrig, Günter Hommel, Klaus-Peter Löhr, Bernd Mahr, Peter Pepper and Radu Popescu-Zeletin for establishing this institution, for the educational program within the Graduiertenkolleg and their ongoing support. Also, I would like to thank the other fellows being supported within the Graduiertenkolleg, especially Andreas Polze for discussions on our Ph.D. projects.

The Graduiertenkolleg is supported by the Deutsche Forschungsgemeinschaft DFG and provided the author with a three year grant. I am thankful for this provision of a material basis for my research, which also included support for the presentation of my work at conferences.

My work has been stimulated in the beginning by discussions with the members of the Projektinitiative Medizin-Informatik (PMI), headed by Horst Hansen who introduced me to the context of a real-life project of designing and implementing an open system in a medical environment.

The members of the department Funktionales und Logisches Programmieren provided me with a pleasurable an pleasing environment to perform my research. Especially, I would like to thank Dirk Lutzebäck for ongoing and valuable discussions on open systems and ODP.

Professor Peter Löhr helped to increase the quality of this thesis with detailed comments and corrections.

Finally, and most of all, I would like to express my thanks to my supervisor, professor Bernd Mahr, for his continuous support which nearly always set me on a “right track” to follow in my work, for his stimulating and most important advice in structuring this thesis, and for his confidence which gave me more than enough room to follow my research interests in a independent way which I have to doubt to have again in the future.

Appendix A

Laura's subtyping tested by Alice-agents

Every implementation of a coordination language such as Laura necessarily involves performing computations such as determining subtype-relations for service-interfaces. This computation then requires coordination in itself, for example to implement a concurrent algorithm for subtype-testing.

In chapter 3 we introduced Alice as a coordination assembler. In this appendix we demonstrate how Alice can be used to implement Laura's rules for subtyping of interfaces. This results in a concurrent algorithm opposed to the sequential C-routines we use in our experimental prototype.

Let the following service interfaces offer an operation to purchase a travel-ticket for families. The first gets some information about seating preferences such as a row-number and information about the number of children an whether a swaddling-desk has to be available:

family-a=

```
(familyticket : seating * childinfo -> ack * cashed)
where
 seating   = <number,character>;
 childinfo = <number,boolean>;
 ack       = boolean;
 cashed    = <number,number>.
```

The second requires less data as arguments. Applying Laura's subtyping-rules shows that it is a subtype of the first.

family-b=

```
(familyticket : seat * children -> ack * cashed)
where
 seat      = <character>;
 childinfo = <number>;
 ack       = boolean;
 cashed    = <number,number>.
```

In order to use ALICE for the implementation of a subtype-algorithm, the interfaces have to be encoded as tuples. If we look at the arguments from *family-a*, they would be encoded as $\langle\langle\perp_{\text{number}},\perp_{\text{character}}\rangle,\langle\perp_{\text{number}},\perp_{\text{boolean}}\rangle\rangle$.

A.1 Testing records

In order to test if two such encodings R1 and R2 fulfill the subtype-relation on anonymous records, we set up a local agent-space in which the fields of R1 are spread and agents resulting from a spreading of R2 try to retrieve fields with in that are in a subtype relation. If all of them terminate, then for all fields of R2 a corresponding field from R1 exists.

We can distinguish three kinds of tuples resulting from the spreading of a tuple that represents a records-type: (1) tuples containing one field of a simple type, called *simple tuples*, (2) tuples containing several fields of simple types, called *flat tuples* and (3) tuples that contain fields of simple types and tuples, called *nested tuples*. The initial idea of subtype-testing in ALICE explained above can be applied to the first two cases.

The corresponding agents are shown in figure A.1. The agent that tests for the subtype relation for anonymous records, r-subtype, has to be initialized with the two records encoded in tuples and two processes-definitions. According to the result of the test, one of them will be executed as an agent, using the bool-cond meta-agents. r-subtype starts the test by executing test-simple in a local agent-space.

r-subtype:	$\langle\!\langle\{\text{a,b,p,q}\}\text{,,in(c:}\{\langle\text{test-simple}\rangle_{\{\text{a,b}\}}\}\text{).out}(\langle\text{c,p,q}\rangle)\rangle\!\rangle$
test-simple:	$\langle\!\langle\{\text{a,b}\}\text{,,in(c:}\{\langle\text{b}\rangle,\langle\text{in}(\langle\perp_{\text{Simple}}\rangle)\rangle\}\text{).}$
	$\text{out}(\langle\text{c,}\langle\text{match-simple}\rangle_{\{\text{a,b}\}},\langle\text{test-flat}\rangle_{\{\text{a,b}\}}\rangle)\rangle\!\rangle$
match-simple:	$\langle\!\langle\{\text{a,b}\}\text{,,in(c:}\{\langle\text{a}\rangle,\langle\text{in}(\langle\text{b}\rangle)\rangle\}\text{).out}(\langle\text{c,,block}\rangle)\rangle\!\rangle$
test-flat:	$\langle\!\langle\{\text{a,b}\}\text{,,in(c:}\{(\!\vert\text{b}\vert\!),(\!\vert\text{in}(\langle\perp_{\text{Simple}}\rangle)\vert\!)_{\{\text{b}\}}\}\text{).}$
	$\text{out}(\langle\text{c,}\langle\text{match-flat}\rangle_{\{\text{a,b}\}},\langle\text{match-nested}\rangle_{\{\text{a,b}\}}\rangle)\rangle\!\rangle$
match-flat:	$\langle\!\langle\{\text{a,b}\}\text{,,in(c:}\{(\!\vert\text{a}\vert\!),(\!\vert\text{in}(\langle\text{b}\rangle)\vert\!)\}\text{).out}(\langle\text{c,,block}\rangle)\rangle\!\rangle$

Figure A.1: Agents for the testing of simple and flat tuples

This agent first tests in a local agent-space, if b is a simple tuple with a single field only. If this holds, match-simple is started, otherwise test-flat. Note that the test on b suffices as then a has to be a simple tuple, too. If it is not, match-simple will fail.

match-simple easily tests the subtype relation for simple tuples by putting the supposed subtype tuple together with an agent that in's the field from the supposed supertype in a local agent-space. As simple tuples have to be identical, ALICE's matching can be applied. If the tuples do not match, the meta-agent block is issued which is defined to be never terminating. In this case, the local agent-space of r-subtype results in a $\langle$F$\rangle$.

test-flat first tests by spreading the tuple b and a number of agents that try to in a simple tuple. If this holds, match-flat tests the subtype relation in a similar way to match-simple. Again, if a is not a flat tuple, match-flat will fail so that the test on b suffices.

In match-flat, a is spread in a local agent-space together with agents resulting from spreading b. If all of these agents terminate, then they all found a matching tuple resulting from the spreading of a, which means that the subtype relation holds.

Testing nested tuples is not that straightforward. The problem is to find a combination of fields from both tuples for which the fields are in subtype relations. The ALICE-agents in figure A.2 take the following approach: All possible combinations are generated and given to agents that test the subtype relations for pairs of fields. If one of them succeeds, at least one combination exists for which the subtype relation for records holds.

match-nested: $\langle\!\langle$\{a,b\},,in(c:\{$(\!|\rangle_{\{a\}}$,$(\!|$in($\langle\rangle$)$|\!)_{\{b\}}$\}).
out($\langle$c,$\langle$blow-up$\rangle_{\{a,b\}}$,block$\rangle$).out($\langle$in($\langle\rangle$)$\rangle$)$\rangle\!\rangle$

blow-up: $\langle\!\langle$\{a,b\},,in(c:\{[$\langle\rangle$|a],$\langle$in([$\langle\rangle$|b])$\rangle$\}).
out($\langle$c,$\langle$shuffle$\rangle_{\{a,b,b\}}$,$\langle$blow-up$\rangle_{\{a,\langle(\!|b|\!),\langle\rangle\rangle\}}\rangle\rangle\!\rangle$

shuffle: $\langle\!\langle$\{a,b,e\},,in(c:\{e,$\langle$in($\langle\rangle$)$\rangle$\}).out($\langle$c,,$\langle$combine-first$\rangle_{\{a,b\}}\rangle$).
out($\langle$c,,$\langle$shuffle$\rangle_{\{a,\langle|\}b\{|\rangle,\langle(\!|b|\!)\rangle,\langle|\}e\{|\rangle\rangle\}}\rangle\rangle\!\rangle$

combine-first: $\langle\!\langle$\{a,b\},,in(c:\{$\langle$test-pair$\rangle_{\{\langle(\!|a|\!)\rangle,\langle(\!|b|\!)\rangle\}}$\}).
out($\langle$c,,$\langle$combine-rest$\rangle_{\{\langle|\}a\{|\rangle,\langle|\}b\{|\rangle\rangle\}}\rangle\rangle\!\rangle$

combine-rest: $\langle\!\langle$\{a,b\},,in(c:\{a,$\langle$in($\langle\rangle$)$\rangle$\}).out($\langle$c,$\langle$out($\langle\rangle$)$\rangle$,$\langle$shuffle$\rangle_{\{a,b,b\}}\rangle\rangle\!\rangle$

test-pair: $\langle\!\langle$\{a,b\},,in(c:\{b,$\langle$in($\langle\rangle$)$\rangle$\}).out($\langle$c,,$\langle$r-subtype$\rangle_{\{a,b,,block\}}\rangle\rangle\!\rangle$

Figure A.2: Testing nested tuples

match-nested first tests in a local agent-space, if the supposed subtype a has at least as many fields as the supposed supertype b. If it has, the generation of combinations can start. First, blow-up extends b with empty fields until it has the same length as a. r-subtype emits a second agent that tries to in an empty tuple. This tuple plays the role of a signal for the detection of a valid combination. If none is possible, this one agent will block, so that a local agent-space in which r-subtype is executed will evaluate to $\langle$F$\rangle$.

shuffle generates a set of combinations of the fields from a and b. It does so by generating a combine-first agent for a combination and then emits itself with b rotated one field. e plays the role of a counter-field: It is empty, when b has been rotated completely.

combine-first takes the first fields from the combination it is initialized with and tests them in a local agent-space for their subtype-relation with the test-pair agents.

If this test succeeds, then the combinations of the remaining fields have to be tested by combine-rest. Otherwise the combination cannot be one that proved the subtype relation of the two nested tuples and no further action is taken.

combine-rest can emit an agent that emits the empty tuple to signal a valid combinations. Otherwise the remaining fields have to be combined by the shuffle agent. This indirect recursion of shuffle ensures that all combinations of a and b are tested.

test-pair first tests, if b is an empty field. If this holds, then this field was generated by blow-up and the field from a is one of those added to the supertype. In this case, no further action has to be taken. Otherwise a r-subtype agent is emitted to test for the subtype relation of the two fields that blocks if this test fails.

A.2 Testing operation signatures and interfaces

For function types, LAURA defines a contra-variant subtyping which ensures substitutability of a subtype for its supertype. Let a function type be encoded as ⟨⟨argument-types⟩,⟨result-type⟩⟩. The agent o-subtype in figure A.3 tests for the subtype relation amongst such tuples a and b and emits p or q as agents depending on if the test succeeds.

o-subtype:	⟪{a,b,p,q},,in(c:{⟨r-subtype⟩$_{\{\langle\lfloor a\rfloor\rangle,\langle\lfloor b\rfloor\rangle,,\text{block}\}}$,⟨r-subtype⟩$_{\{\langle\langle b\rangle,\langle\langle a\rangle,,\text{block}\}}$}). out(⟨c,p,q⟩)⟫
i-subtype:	⟪{a,b,p,q},,in(c:{(a),(⟨get-op⟩$_{\{b\}}$)}).out(⟨c,p,q⟩)⟫
get-op:	⟪{a},,in(⟨⟨a⟩,b:⊥$_{\text{Tuple}}$⟩).out(⟨o-subtype⟩$_{\{\langle\lfloor b\rfloor\rangle,\langle\lfloor a\rfloor\rangle,,\text{block}\}}$)⟫

Figure A.3: Testing operation signatures and interfaces

o-subtype uses a local agent-space to test if arguments and results are in the correct subtype relations. It does so by filling it with two appropriately initialized r-subtype agents. If the arguments or results are not in a subtype-relation, the block-agent is emitted, in which case the local agent evaluates to ⟨F⟩.

For service interfaces in LAURA consist of names operation signatures. An interface a is a subtype of an interface b, if for every operation signature from b there is one in a with the same name and its type is a subtype of the one from b.

In ALICE an interface is encoded as a tuple of the form ⟨⟨operation name$_0$,⟨operation type$_0$⟩⟩,..., ⟨operation name$_n$,⟨operation type$_n$⟩⟩⟩. The agent i-subtype in figure A.3 tests for subtyping amongst such tuples a and b and emits p or q as agents depending on if the test succeeds.

i-subtype spreads all operation-signatures from a in a local agent-space together with a set of get-op agents, each initialized with one operation from b. get-op tries to in an operation with the same name. If there is none, the agent blocks. Otherwise the operation-signatures can be tested for the subtype relation with the o-subtype agent, that is initialized to block for the case that the relation does not hold. If all get-op agents terminate, the local agent-space of i-subtype evaluates to ⟨T⟩.

A.3 Example executions

(1) $\Big\{ \langle \text{in}(\langle\rangle)\rangle,$
$\langle \text{shuffle} \rangle_{\{\langle\langle \perp_{\text{number}}, \perp_{\text{character}}\rangle, \langle \perp_{\text{number}}, \perp_{\text{boolean}}\rangle\rangle, \langle\langle \perp_{\text{number}}\rangle, \langle \perp_{\text{character}}\rangle\rangle, \langle\langle \perp_{\text{number}}\rangle, \langle \perp_{\text{character}}\rangle\rangle\}} \Big\} \longrightarrow^*$

(2) $\Big\{ \langle \text{in}(\langle\rangle)\rangle, \langle \text{combine-first} \rangle_{\{\langle\langle \perp_{\text{number}}, \perp_{\text{character}}\rangle, \langle \perp_{\text{number}}, \perp_{\text{boolean}}\rangle\rangle, \langle\langle \perp_{\text{number}}\rangle, \langle \perp_{\text{character}}\rangle\rangle\}},$
$\langle \text{shuffle} \rangle_{\{\langle\langle \perp_{\text{number}}, \perp_{\text{character}}\rangle, \langle \perp_{\text{number}}, \perp_{\text{boolean}}\rangle\rangle, \langle\langle \perp_{\text{character}}\rangle, \langle \perp_{\text{number}}\rangle\rangle, \langle\langle \perp_{\text{character}}\rangle\rangle\}} \Big\} \longrightarrow^*$

(3) $\Big\{ \langle \text{in}(\langle\rangle)\rangle, \langle \text{combine-rest} \rangle_{\{\langle\langle \perp_{\text{number}}, \perp_{\text{boolean}}\rangle\rangle, \langle\langle \perp_{\text{character}}\rangle\rangle\}},$
$\langle \text{combine-first} \rangle_{\{\langle\langle \perp_{\text{number}}, \perp_{\text{character}}\rangle, \langle \perp_{\text{number}}, \perp_{\text{boolean}}\rangle\rangle, \langle\langle \perp_{\text{character}}\rangle, \langle \perp_{\text{number}}\rangle\rangle\}},$
$\langle \text{shuffle} \rangle_{\{\langle\langle \perp_{\text{number}}, \perp_{\text{character}}\rangle, \langle \perp_{\text{number}}, \perp_{\text{boolean}}\rangle\rangle, \langle\langle \perp_{\text{number}}\rangle, \langle \perp_{\text{character}}\rangle\rangle, \langle\rangle\}} \Big\} \longrightarrow^*$

(4) $\Big\{ \langle \text{in}(\langle\rangle)\rangle, \langle \text{shuffle} \rangle_{\{\langle\langle \perp_{\text{number}}, \perp_{\text{boolean}}\rangle\rangle, \langle\langle \perp_{\text{character}}\rangle\rangle, \langle\langle \perp_{\text{character}}\rangle\rangle\}},$
$\langle \text{combine-rest} \rangle_{\{\langle\langle \perp_{\text{number}}, \perp_{\text{boolean}}\rangle\rangle, \langle\langle \perp_{\text{number}}\rangle\rangle\}} \Big\} \longrightarrow^*$

(5) $\Big\{ \langle \text{in}(\langle\rangle)\rangle, \langle \text{combine-first} \rangle_{\{\langle\langle \perp_{\text{number}}, \perp_{\text{boolean}}\rangle\rangle, \langle\langle \perp_{\text{character}}\rangle\rangle\}},$
$\langle \text{shuffle} \rangle_{\{\langle\langle \perp_{\text{number}}, \perp_{\text{boolean}}\rangle\rangle, \langle\langle \perp_{\text{character}}\rangle\rangle, \langle\rangle\}}$
$\langle \text{shuffle} \rangle_{\{\langle\langle \perp_{\text{number}}, \perp_{\text{boolean}}\rangle\rangle, \langle\langle \perp_{\text{number}}\rangle\rangle, \langle\langle \perp_{\text{number}}\rangle\rangle\}} \Big\} \longrightarrow^*$

(6) $\Big\{ \langle \text{in}(\langle\rangle)\rangle, \langle \text{combine-first} \rangle_{\{\langle\langle \perp_{\text{number}}, \perp_{\text{boolean}}\rangle\rangle, \langle\langle \perp_{\text{number}}\rangle\rangle\}},$
$\langle \text{shuffle} \rangle_{\{\langle\langle \perp_{\text{number}}, \perp_{\text{boolean}}\rangle\rangle, \langle\langle \perp_{\text{number}}\rangle\rangle, \langle\rangle\}} \Big\} \longrightarrow^*$

(7) $\Big\{ \langle \text{in}(\langle\rangle)\rangle, \langle \text{combine-rest} \rangle_{\{\langle\rangle, \langle\rangle\}} \Big\} \longrightarrow^*$

(8) $\Big\{ \langle \text{in}(\langle\rangle)\rangle, \langle \text{out}(\langle\rangle)\rangle \Big\} \longrightarrow^*$

(9) $\Big\{ \langle \text{in}(\langle\rangle)\rangle, \langle\rangle \Big\} \longrightarrow^*$

(10) $\Big\{ \Big\}$

Figure A.4: An example of generating the permutations

In this appendix we illustrate an execution of the r-subtype agent from section A.1. Figure A.4 shows snapshots from the local agent-space of r-subtype that result from executing match-nested with the tuples a=$\langle\langle \perp_{\text{number}}, \perp_{\text{character}}\rangle, \langle \perp_{\text{number}}, \perp_{\text{boolean}}\rangle\rangle$ and b=$\langle\langle \perp_{\text{number}}\rangle, \langle \perp_{\text{character}}\rangle\rangle$. The snapshots taken are only one possible execution.

In snapshot (2) combine-first starts to test the first combination $\langle\langle \perp_{\text{number}}, \perp_{\text{character}}\rangle\rangle$ from a and $\langle\langle \perp_{\text{number}}\rangle\rangle$, two flat tuples that are in the subtype relation. The resulting

combine-rest (4) leads to the combine-first in (5) where the test for subtyping fails. In (7) the combine-rest results from the successful testing of the other permutation of the fields from a and b. It out's the agent in (8) that emits the empty signal tuple. In (10) all agents that tested have terminated, including the in($\langle\rangle$)-agent that terminated because one combination reflected that a$\lessdot$ b holds. The local agent-space of r-subtype therefore evaluates to $\langle$T$\rangle$.

Appendix B

Bibliography

[Agha and Callsen, 92] Gul Agha and Christian J. Callsen. ActorSpaces: An Open Distributed Programming Paradigm. Technical Report UIUCDCS-R-92-1766, University of Illinois at Urbana-Champaign, 1992.

[Agha and Callsen, 93] G. Agha and C. Callsen. ActorSpaces: An Open Distributed Programming Paradigm. In *Proceeding of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[Ahmed and Gelernter, 91a] Shakil Ahmed and David Gelernter. A Higher-Level Environment for Parallel Programming. Technical Report YALEU/DCS/RR-877, Yale University, 1991.

[Ahmed and Gelernter, 91b] Shakil Ahmed and David Gelernter. Program Builders as Alternatives to High-Level Languages. Technical Report YALEU/DCS/RR-887, Yale University, 1991.

[Aho and Sethi et al, 86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, 1986.

[Ahuja and Carriero et al, 88] Sudhir Ahuja, Nicholas J. Carriero, David H. Gelernter, and Venkatesh Krishnaswamy. Matching Language and Hardware for Parallel Computation in the Linda Machine. *IEEE Transactions on Computers*, 37(8):921–929, 1988.

[Amadio and Cardelli, 91] Roberto M. Amadio and Luca Cardelli. Subtyping Recursive Types. In *Proceedings of the 18th annual ACM Symposium on Principles of Programming Languages*, pages 104–118, 1991.

[Anderson and Shasha, 91] Brian G. Anderson and Dennis Shasha. Persistent Linda: Linda + Transactions + Query Processing. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, number 574 in LNCS, pages 93–109. Springer, 1991.

[Andreoli and Ciancarini et al, 92a] Jean-Marc Andreoli, Paolo Ciancarini, and Remo Pareschi. Interaction Abstract Machines. Technical Report 92-23, ECRC, 1992.

[Andreoli and Ciancarini et al, 92b] JM Andreoli, P. Ciancarini, and R. Pareschi. Parallel Searching with Multisets-as-Agents. Technical Report TR 30/92, University of Pisa, 1992.

[Andreoli and Pareschi, 91] Jean-Marc Andreoli and Remo Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.

[Baier and Majster-Cederbaum, 94] Christel Baier and Mila E. Majster-Cederbaum. The connection between an event structure semantics and an operational semantics for *TCSP*. *Acta Informatica*, 31:81–104, 1994.

[Bakken and Schlichting, 91] David E. Bakken and Richard D. Schlichting. Tolerating failures in the bag-of-tasks programming paradigm. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, pages 248–255, 1991.

[Bakken and Schlichting, 93] David E. Bakken and Richard D. Schlichting. Supporting Fault-Tolerant Parallel Programming in Linda. Technical Report TR 93-18, Department of Computer Science, The University of Arizona, 1993.

[Banâtre and Le Métayer, 91] Jean-Pierre Banâtre and Daniel Le Métayer. Introduction to Gamma. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, number 574 in LNCS, pages 197–202. Springer, 1991.

[Banâtre and Le Métayer, 93] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, 1993.

[Beguelin and Dongarra et al, 91] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam. *A Users' Guide to PVM Parallel Virtual Machine*. Oak Ridge National Laboratory, 1991.

[Berndt, 89] D. Berndt. *C-Linda reference Manual (DRAFT) Beta Version 2.0*. Scientific Research Associates, 1989.

[Berry and Boudol, 90] Gérard Berry and Gérard Boudol. The Chemical Abstract Machine. In *Proceeding of the 17th ACM Symposium on Principles of Programming Languages*, pages 81–94, 1990.

[Birman, 91] Kenneth P. Birman. The Process Group Approach to Reliable Distributed Computing. Technical Report TR91-1216, Department of Computer Science, Cornell University, 1991.

[Birman and Cooper et al, 90] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makgangou, K. Kane, F. Schmuck, and M. Wood. *The ISIS System Manual, Version 2.1.* The ISIS Project, 1990.

[Birman and Schiper et al, 91] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[Bjornson, 87] R. Bjornson. *A Linda User's Manual.* Scientific Computing Associates, 1987.

[Bjornson and Carriero et al, 88] Robert Bjornson, Nicholas Carriero, David Gelernter, and Jerrold Leichter. Linda, the Portable Parallel. Technical Report YALE/DCS/RR-520, Yale University, 1987, revised 1988.

[Borrmann and Herdieckerhoff, 88] Lothar Borrmann and Martin Herdieckerhoff. Linda integriert in Modula-2 – ein Sprachkonzept für portable parallele Software. In U. Kastens and F.J. Rammig, editors, *Architektur und Betrieb von Rechensystemen*, pages 106–118. GI/ITG, 1988.

[Bowman and Debray et al, 93] Mic Bowman, Saumya K. Debray, and Larry L. Peterson. Reasoning About Naming Systems. *ACM Transactions on Programming Languages and Systems*, 15(5):795–825, 1993.

[Broadbery and Playford, 91] Peter Broadbery and Keith Playford. Using Object-Oriented Mechanisms to Describe Linda. In Greg Wilson, editor, *Linda-Like Systems and Their Implementation*, pages 14–26. Edinburgh Parallel Computing Centre, 1991. Technical Report 91-13.

[Butcher, 91] Paul Butcher. Lucinda. In Greg Wilson, editor, *Linda-Like Systems and Their Implementation*, pages 27–38. Edinburgh Parallel Computing Centre, 1991. Technical Report 91-13.

[Butcher and Zedan, 91a] Paul Butcher and Hussein Zedan. Lucinda – A Polymorphic Linda. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, number 574 in LNCS, pages 126–146. Springer, 1991.

[Butcher and Zedan, 91b] Paul Butcher and Hussein Zedan. Lucinda – An Overview. *ACM SIGPLAN Notices*, 26(8):90–100, 1991.

[Butler and Leveton et al, 93] Ralph M. Butler, Alan L. Leveton, and Ewing L. Lusk. *p4-Linda: A Portable Implementation of Linda*, 1993.

[Butler and Lusk, 92] Ralph Butler and Ewing Lusk. User's guide to the p4 parallel programming system. Technical Report ANL-92/17, Argonne National Laboratory, Mathematics and Computer Science Division, 1992.

[Callahan and Purtilo, 90] John R. Callahan and James M. Purtilo. A Packaging System for Heterogeneous Execution Environments. Technical Report 2542, University of Maryland CSD, 1990.

[Callsen and Cheng et al, 91] Christian J. Callsen, Ivan Cheng, and Per L. Hagen. The AUC C++ Linda System. In Greg Wilson, editor, *Linda-Like Systems and Their Implementation*, pages 39–73. Edinburgh Parallel Computing Centre, 1991. Technical Report 91-13.

[Cardelli, 88] Luca Cardelli. Structural Subtyping and the Notion of Power Type. In *Proceedings of the 15th annual ACM Symposium on Principles of Programming Languages*, pages 70–79, 1988.

[Carriero and Gelernter, 86] Nicholas Carriero and David Gelernter. The S/Net's Linda Kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, 1986.

[Carriero and Gelernter, 88] Nicholas Carriero and David Gelernter. Applications Experience with Linda. In *Proceedings of the ACM/SIGPLAN PPEALS 1988*, pages 173–187, 1988.

[Carriero and Gelernter, 89a] Nicholas Carriero and David Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, 1989.

[Carriero and Gelernter, 89b] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.

[Carriero and Gelernter, 89c] Nicholas Carriero and David Gelernter. Tuple analysis and partial evaluation strategies in the Linda precompiler. In *Proceedings of the 2nd Workshop on Languages and Compilers for Parallelism*, 1989.

[Carriero and Gelernter, 91] Nicholas Carriero and David Gelernter. New Optimization Strategies for the Linda Pre-Compiler. In Greg Wilson, editor, *Linda-Like Systems and Their Implementation*, pages 74–83. Edinburgh Parallel Computing Centre, 1991. Technical Report 91-13.

[Carriero and Gelernter et al, 94] Nicholas Carriero, David Gelernter, and Leonore Zuck. Bauhaus-Linda. In *Workshop on Languages and Models for Coordination, European Conference on Object Oriented Programming*, 1994.

[Carroll, 60] Lewis Carroll. *The Annotated Alice – Alice's Adventures in Wonderland & Through the Looking Glass.* Penguin Books, 1960. Edited by Martin Gardner.

[Chiba and Kato et al, 91] Shigeru Chiba, Kazuhiko Kato, and Takashi Masuda. Exploiting a Weak Consistency to Implement Distributed Tuple Space. In *Proceedings of the 12th IEEE International Conference on Distributed Computing Systems ICDCS 92*, pages 416–423, 1991.

[Cho89] Chorus Supercomputer Inc, New York. *Linda-C Documentation*, 1989.

[Ciancarini, 93] Paolo Ciancarini. Coordinating Rule-Based Software Processes with ESP. *ACM Transactions on Software Engineering*, 2(3):203–227, 1993. Also as technical report UBLCS-93-8, University of Bologna.

[Ciancarini and Guerrini, 93] P. Ciancarini and N. Guerrini. Linda meets Minix. *Operating Systems Reviews*, 27(4):76–92, 1993.

[Ciancarini and Jensen et al, 92] Paolo Ciancarini, Keld K. Jensen, and Dani Yankelevich. The Semantics of a Parallel Language based on a Shared Dataspace. Technical Report TR 26/92, University of Pisa, 1992.

[Cog89a] Cogent Research Inc. *Process creation in QIX*, 1989. Technical Note 89.3.

[Cog89b] Cogent Research Inc., Beaverton, Oregon. *XTM Product Specification*, 1989.

[Cog90] Cogent Research Inc. *Kernel Linda Specification - Version 4.0*, June 1990. Technical Note 89.17.

[Dai, 88] Kechang Dai. *Large-Grain Dataflow Computation and Its Architectural Support*. PhD thesis, TU Berlin, 1988.

[Dai and Giloi, 90a] Kechang Dai and Wolfgang K. Giloi. A Basic Architecture Supporting LGDG Computation. In *Proceedings of the International Conference on Supercomputing*, pages 23–33, 1990.

[Dai and Giloi, 90b] Kechang Dai and Wolfgang K. Giloi. A Non-Branch RISC Kernel for Large-Grain Dataflow Computations. In *Proceedings of the V. International Workshop on Parallel Processing by Cellular Automata and Arrays, Berlin*, pages 183–188, 1990.

[Darlington and Reeve, 81] J. Darlington and M. Reeve. ALICE: A MultiProcessor Reduction Machine for thhe Parallel Evaluation of Applicative Languages. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, 1981.

[De Nicola and Pugliese, 93] Rocco De Nicola and Rosario Pugliese. Testing Linda: Observational Semantics for an Asynchronous Language. Technical report, 1993.

[Donnelly and Stallman, 92] Charles Donnelly and Richard Stallman. *Bison – The YACC-compatible Parser Generator, Bison Version 1.20*, December 1992.

[Ehrig and Mahr, 85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1.* EATACS Monographs. Springer, 1985.

[Gayda, 92] Christian Gayda. *DOKMA Ein Dokumentenorientiertes Kommunikationsmodell für medizinische Anwendungen als Basis für ein Rollensystem im medizinischen Umfeld.* PhD thesis, TU-Berlin, 1992. In German.

[Gelernter, 82] D. Gelernter. *An integrated microcomputer network for experiments in distributed programming.* PhD thesis, SUNY at Stony Brooks, 1982.

[Gelernter, 85] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[Gelernter, 89] David Gelernter. Multiple tuple spaces in Linda. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89, Vol. II: Parallel Languages*, LNCS 366, pages 20–27, 1989.

[Gelernter, 91] David Gelernter. *Mirror Worlds.* Oxford University Press, New York, 1991. ISBN 0-19-506812-2.

[Gelernter and Carriero, 92] David Gelernter and Nicholas Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–107, 1992.

[Gelernter and Philbin, 90] David Gelernter and James Philbin. Spending Your Free Time. *BYTE*, 15(5):213–219, 1990.

[Hansen and Kutsche et al, 91] Horst Hansen, Ralf-Detlef Kutsche, and Joachim Steffens. The PADKOM System Model – an Open Platform for Medical Applications in a Distributed Multimedia Environment. In Jan de Meer and Volker Heymer, editors, *Proceedings of the International IFIP Workshop on Open Distributed Processing*, 1991.

[Hansen and Kutsche, 94] H. Hansen and R.-D. Kutsche. Medical Applications of ODP. In J. de Meer, B. Mahr, and S. Storp, editors, *Proceedings of the International IFIP Conference on Open Distributed Processing*, pages 67–99. North-Holland, 1994.

[Hasselbring, 91] W. Hasselbring. Combining SETL/E with Linda. In Greg Wilson, editor, *Linda-Like Systems and Their Implementation*, pages 84–99. Edinburgh Parallel Computing Centre, 1991. Technical Report 91-13.

[Hasselbring, 92] W. Hasselbring. A Formal Z Specification of ProSet-Linda. Technical Report 04.92, University of Essen, 1992.

[Hasselbring, 93a] W. Hasselbring. Formale Spezifikation und Prototyping im Sprachentwurf: Eine Fallstudie. Technical Report 05–92, University of Essen, 1993. Also in Proceedings of GI–Fachtagung Softwaretechnik, Dortmund, Germany, 1993.

[Hasselbring, 93b] W. Hasselbring. Prototyping Parallel Algorithms with ProSet-Linda. In Jens Volkert, editor, *Proceedings of the 2nd International Conference of the Austrian Center for Parallel Computation*, number 734 in LNCS, pages 135–150. Springer, 1993. Also as [Hasselbring, 93c].

[Hasselbring, 93c] W. Hasselbring. Prototyping Parallel Algorithms with ProSet-Linda. Technical Report 04–92, University of Essen, 1993. Also as [Hasselbring, 93b].

[Hennessy, 90] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics.* Wiley Press, 1990.

[Hopcroft and Ullman, 79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation.* Addison-Wesley, 1979.

[Hupfer and Kaminsky et al, 91] Susanne Hupfer, David Kaminsky, Nicholas Carriero, and David Gelernter. Coordination Applications of Linda. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, number 574 in LNCS, pages 187–194. Springer, 1991.

[IBM93] IBM Corp. *SOMobjects Developer Toolkit – Technical Overview, Version 2.0*, November 1993.

[ISO, 91] ISO. Quality management and quality system elements – Part 2: Guidelines for services, 1991. International standard ISO 9004-2.

[ISO/IEC JTC1/SC21, 93] ISO/IEC JTC1/SC21. Information Technology – Open Distributed Processing – ODP Trading Function, WG7 Working Document, 1993.

[ISO/IEC JTC1/SC21/WG7, 91] ISO/IEC JTC1/SC21/WG7. Basic Reference Model of Open Distributed Processing – Part 5: Architectural Semantics, Working Document, 1991.

[ISO/IEC JTC1/SC21/WG7, 93a] ISO/IEC JTC1/SC21/WG7. Basic Reference Model of Open Distributed Processing – Part 1-4, 1993.

[ISO/IEC JTC1/SC21/WG7, 93b] ISO/IEC JTC1/SC21/WG7. Basic Reference Model of Open Distributed Processing – Part 1: Overview, Working Draft, 1993.

[ISO/IEC JTC1/SC21/WG7, 94a] ISO/IEC JTC1/SC21/WG7. Basic Reference Model of Open Distributed Processing – Part 2: Descriptive Model, Draft International Standard, 1994. ISO/IEC DIS 10746-2, ITU-T Draft Rec. X.902.

[ISO/IEC JTC1/SC21/WG7, 94b] ISO/IEC JTC1/SC21/WG7. Basic Reference Model of Open Distributed Processing – Part 3: Prescriptive Model, Draft International Standard, 1994. ISO/IEC DIS(E) 10746-3, ITU-T Rec. X.903.

[Jagannathan, 90] Suresh Jagannathan. Semantics and Analysis of First-Class Tuple Spaces. Technical Report YALEU/DCS/RR-783, Yale University, 1990.

[Jagannathan, 91] Suresh Jagannathan. Expressing Fine-Grained Parallelism Using Concurrent Data Structure. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, number 574 in LNCS, pages 77–92. Springer, 1991.

[Jellinghaus, 90] Robert Jellinghaus. Eiffel Linda: An Object-Oriented Linda Dialect. *ACM SIGPLAN Notices*, 25(12):70–84, 1990.

[Jensen and Riksted, 89] Keld K. Jensen and Gorm E. Riksted. Linda – A Distributed Programming Paradigm. Master's thesis, University of Aalborg, 1989.

[Jopp, 94] Klaus Jopp. Wem gehört dieser Koffer? *Die Zeit*, page 26, July, 22 1994. In German.

[Kane, 91] A. J. Kane. A Simple Linda-C Parallel Processing Environment For Symmetric Multi-processing VAX/VMS Systems. Master's thesis, East Tennessee State University, 1991.

[Kaplan and Love et al, 92] Simon Kaplan, Christopher Love, Alan M. Carroll, and Daniel M. LaLiberte. *Epoch 4.0, a modified version of GNU Emacs*, 1992.

[Kornfeld, 79] Willian A. Kornfeld. ETHER – A Parallel Problem Solving System. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence IJCAJ, Tokyo*, 1979.

[Krishnaswamy and Ahuja et al, 88] Venkatesh Krishnaswamy, Sudhir Ahuja, Nicholas J. Carriero, and David Gelernter. The Architecture of a Linda Coprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 240–249, 1988.

[Kutsche, 94] Ralf-Detlef Kutsche. *A type-oriented approach to the specification and formal semantics of a distributed, heterogeneous object system.* PhD thesis, TU-Berlin, 1994.

[Leler, 90] Wm Leler. Linda Meets Unix. *IEEE Computer*, 23(2):43–54, 1990.

[Loogen and Goltz, 91] Rita Loogen and Ursula Goltz. Modelling nondeterministic concurrent processes with event structures. *Fundamenta Informaticae*, 14:39–73, 1991.

[LRW90] LRW Systems. *VAX* Linda-C *Installation Guide*, August 1990. Order Number VLN-IG-101.

[LRW91a] LRW Systems. *VAX* Linda-C *Release Notes*, April 1991. Order Number VLN-RN-103.

[LRW91b] LRW Systems. *VAX* Linda-C *User's Guide*, September 1991. Order Number VLN-UG-102.

[Mahr, 94] Bernd Mahr. Applications of Type Theory. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proceedings of TAPSOFT'93: Theory and Practice of Software Development*, LNCS 668, pages 343–355. Springer, 1994.

[Mahr and Sträter et al, 90] B. Mahr, W. Sträter, and C. Umbach. Fundamentals of a Theory of Types and Declarations. Technical Report KIT-Report 82, TU-Berlin, 1990.

[Mahr and Tolksdorf, 93] Bernd Mahr and Robert Tolksdorf. Coordination and Logic Programming. In H. Reichel, editor, *Informatik, Wirtschaft, Gesellschaft, Proceedings of 23. GI Jahrestreffen 1993 – Fachgespräch Kooperation und Konkurrenz*, pages 545–550, 1993.

[Marzetta, 92] Markus Marzetta. Universes in the theories of types and names. In E. Börger, G. Jäger, H. Kleine Büning, S. Martini, and M.M. Richter, editors, *Proc. of the 6th Workshop on Computer Science Logic*, LNCS 702, pages 340–351. Springer, 1992.

[Matsuoka, 88] Satoshi Matsuoka. Tuple Space Communication in Distributed Object-Oriented Computing. Master's thesis, University of Tokyo, 1988.

[Matsuoka and Kawai, 88] Satoshi Matsuoka and Satoru Kawai. Using Tuple Space Communication in Distributed Object-Oriented Languages. In *Conference Proceedings OOPSLA '88*, pages 276–284, 1988.

[Matthews, 88] Stuart R. Matthews. The Specification and Design of a Nondeterministic Data Structure Using CCS. In C. Rattay, editor, *Specification and Verification of Concurrent Systems*, pages 500–525. Springer and British Computer Society, 1988.

[Mattson and Bjornson et al, 92] Timothy G. Mattson, Rob Bjornson, and David Kaminsky. The C-Linda Language for Networks of Workstations. In *Workshop on Cluster Computing, Florida State Universit*, 1992.

[Monro, 87] G.P. Monro. The Concept of Multiset. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 33:171–178, 1987.

[Moor, 82] Ian W. Moor. An Applicative Compiler for a Parallel Machine. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 284–293, 1982. ACM SIGPLAN Notices, 17(6).

[Mussat, 91] Louis Mussat. Parallel Programming with Bags. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, number 574 in LNCS, pages 203–218. Springer, 1991.

[Myers and Purtilo, 92] Heidi E. Myers and James M. Purtilo. Interface Type Checking for Large C Applications. *Computer Languages*, 17(2):147–154, 1992.

[Narem Jr., 89] James E. Narem Jr. An Informal Operational Semantics of C-Linda V2.3.5. Technical Report YALEU/DCS/TR-839, Yale University, 1989.

[NeX94] NeXT Computer, Inc. *OpenStep Specification – Summary Version, 6/30/94 Draft*, 1994.

[OMG91] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design, Inc., and SunSoft, Inc. *The Common Object Request Broker: Architecture and Specification*, 1991.

[OMG92] Object Management Group. *OMG Architecture Guide Chapter 4: The OMG Object Model*, 1992.

[Padget and Broadbery et al, 91] Julian Padget, Peter Broadbery, and David Hutchinson. Mixing Concurrency Abstraction and Classes. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, number 574 in LNCS, pages 174–186. Springer, 1991.

[Paxson, 92] Vern Paxson. *FLEX Lexical Analyzer Generator*, 1992.

[Pinakis, 91] James Pinakis. The Inclusion of the Linda Tuple Space Operations in a Pascal-based Concurrent Language. In *Proceedings of the 14th Australian Computer Science Conference*, 1991.

[Polze, 93] Andreas Polze. The Object Space Approach: decoupled communication in C++. In *Proceedings of TOOLS USA'93*, 1993.

[Polze, 94] Andreas Polze. *Objektorientierung und lose gekoppelte Kommunikation als Basis für die Entwicklung offener, verteilter Anwendungssysteme.* PhD thesis, Freie Universität Berlin, 1994. In German.

[Polze and Löhr, 92] Andreas Polze and Klaus-Peter Löhr. Kommunikationsstructuren in nebenläufigen Systemen und ihre programmiersprachliche Realisierung. Technical report, Institut für Informatik, FU Berlin, 1992. In German.

[Pooyan, 92] Ladan Pooyan. ϵ-Structures as Semantic Models of the Λ-Calculus. Master's thesis, TU-Berlin, 1992.

[Purtilo, 90] James M. Purtilo. The Polylith Software Bus. Technical Report 2469, University of Maryland CSD, 1990.

[Purtilo and Atlee, 91] James M. Purtilo and Joanne M. Atlee. Module Reuse by Interface Adaption. *Software-Practice and Experience*, 21(6):538–555, 1991.

[Purtilo and Jalote, 89] James M. Purtilo and Pankaj Jalote. An Environment for Prototyping Distributed Applications. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 588–594, 1989.

[Raymond, 94] K.A. Raymond. Reference Model of Open Distributed Processing: a Tutorial. In J. de Meer, B. Mahr, and S. Storp, editors, *Proceedings of the International IFIP Conference on Open Distributed Processing*, pages 3–14. North-Holland, 1994.

[Schoinas, 91a] G. Schoinas. *Issues on the implementation of PrOgramming SYstem for distriButed appLications*. University of Crete, 1991.

[Schoinas, 91b] G. Schoinas. POSYBL: Implementing the Blackboard Model in a Distributed Memory Environment Using Linda. In Greg Wilson, editor, *Linda-Like Systems and Their Implementation*, pages 105–116. Edinburgh Parallel Computing Centre, 1991. Technical Report 91-13.

[Seyfarth and Bickham et al, 94] Benjamin R. Seyfarth, Jerry L. Bickham, and Mangaiarkarasi Arumughum. *Glenda Installation and Use*, 1994.

[Shannon and Snodgrass, 89] Karen Shannon and Richard Snodgrass. Mapping the Interface Description Language Type Model into C. *IEEE Transactions on Software Engineering*, 15(11):1333–1346, 1989.

[Soley, 93] Mark Soley, Richard. An object model for integration. *Computer Standards & Interfaces*, 15(2-3):149–166, 1993.

[SRA] Scientific Research Associates. *C-Linda User's Guide & Reference Manual.*

[Sträter, 92] Werner Sträter. ϵ_T*: Eine Logik mit Selbstreferenz und totalem Wahrheitswert.* PhD thesis, Technische Universität Berlin, 1992. In German.

[Subrahmanyam, 81] P. A. Subrahmanyam. Nondeterminism in Abstract Data Types. In S. Evan and O. Kariv, editors, *Automata, Languages and Programming*, LNCS 115, pages 148–164. Springer, 1981.

[SUNa] External Data Representation: Protocol Specficiation.

[SUNb] External Data Representation: Sun Technical Notes.

[Sutcliffe and Pinakis, 90] G. Sutcliffe and J. Pinakis. Prolog-Linda – An Embedding of Linda in muProlog. In C.P. Tsang, editor, *Proceedings of the AI'90 – the 4th Australian Conference on Artificial Intelligence*, pages 331–340, 1990.

[Sutcliffe and Pinakis, 91] Geoff Sutcliffe and James Pinakis. Prolog-D-Linda: An Embedding of Linda in SICStus Prolog. Technical Report 91/7, The University of Western Australia, Department of Computer Science, 1991.

[TRI94] FEST Project Consortium, NUCLEUS Project Consortium, SHINE Project Consortium: Integrated TRILOGY Framework, 1994. AIM-Project A2055.

[Various authors, 89] Various authors. Technical Correspondence on "Linda in Context". *Communications of the ACM*, 32(10):1244–1258, 1989.

[Westbrook and Zuck, 94] Jeffrey Westbrook and Leonore Zuck. Adaptive Algorithms for PASO Systems. In *Proceeding of Conference on Principles of Distributed Computing PODC'94*, 1994.

[Winskel, 88] Glynn Winskel. An introduction to event structures. In J.W. de Bakken, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, pages 364–397. Springer, 1988.

[Wittkugel, 94] Torsten Wittkugel. *Synchronisationsmechanismen in verteilten Objektsystemen.* PhD thesis, TU-Berlin, 1994. In German.

[Yeo and Ananda et al, 93] A.K. Yeo, A.L. Ananda, and E.K. Koh. A Taxonomy of Issues in Name Systems Design and Implementation. *Operating Systems Reviews*, 27(3):4–18, 1993.

[Zenith, 91a] Steven Ericsson Zenith. The Axiomatic Characterization of *Ease*. In Greg Wilson, editor, *Linda-Like Systems and Their Implementation*, pages 143–152. Edinburgh Parallel Computing Centre, 1991. Technical Report 91-13.

[Zenith, 91b] Steven Ericsson Zenith. A Rationale for Programming with *Ease*. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, number 574 in LNCS, pages 147–156. Springer, 1991.

This text was processed using the `chanmis`-style