Prototyping Parallel Algorithms in a Set-Oriented Language

Dissertation

zur Erlangung des Grades des

Doktors der Naturwissenschaften

der Universität Dortmund

am Fachbereich Informatik

von

Wilhelm Hasselbring

Dortmund

1994

Tag der mündlichen Prüfung: 3. Februar 1994

Dekan: Prof. Dr. Peter Marwedel

Gutachter: Prof. Dr. Ernst-Erich Doberkat
Prof. Dr. Franz Stetter

Contents

List of Figures

Abstract

The subject of this thesis is to provide means for prototyping parallel algorithms: if one wants to model an inherently parallel system, it is reasonable to have features for specifying (coarse-grained) processes that communicate and synchronize via a simple communication medium, and not to force such inherent parallelism into sequences. We propose here a new high-level language for prototyping parallel algorithms and applications. Our approach to designing and implementing this tool is somewhat unconventional, as it relies on formal specification and prototyping.

Chapter 1

Introduction

Parallel programming is conceptually harder to undertake and to understand than sequential programming, because a programmer often has to focus on more than one process at a time. Furthermore, programmers are forced on most of today's parallel machines to program on a low level to obtain acceptable performance — ease of use is sacrificed for efficiency. Consequently, developing parallel algorithms is in general considered an awkward undertaking. The goal of our work is to overcome this nuisance by providing a tool for prototyping parallel algorithms.

1.1 Prototyping Parallel Algorithms

Prototyping means constructing a model. Since applications which are inherently parallel should be programmed in a parallel way, it is most natural to incorporate parallelism into the process of model building. Opportunities for automatic detection of parallelism in existing programs are limited (see Sect. 3.1) and furthermore, in many cases the formulation of a parallel program is more natural and appropriate than a sequential one. Most systems in real life are of a parallel nature, thus the intention for integrating parallelism into a prototyping language is not only that of increasing performance. Our intention is to provide means for prototyping parallel algorithms.

As has been observed, no matter how effective the system software and hardware of a parallel machine are at delivering performance, it is only from new algorithms that orders of magnitude improvements in the complexity of a problem can be achieved:

> "An idea that changes an algorithm from n^2 to $n \log n$ operations, where n is proportionate to the number of input elements, is considerably more spectacular than an improvement in machine organization, where only a constant factor of run-time is achieved." [Cocke, 1988, page 250]

Thus, enabling rapid prototyping of parallel algorithms may serve as the basis for developing parallel, high-performance applications.

Most current programming environments for distributed memory architectures, which are usually based on some kind of message passing, provide inadequate support for mapping applications to the machine. In particular, the lack of a global name space forces algorithms to be specified at a relatively low level, since it is complicated to simulate shared memory. This greatly increases the complexity of programs, and also fixes algorithm design choices, inhibiting experimentation with alternate algorithm choices or problem decompositions. This makes programming such machines very difficult, since the programmer has to explicitly encode all the low-level details required to implement the algorithm. The resulting programs are complex and inflexible.

Parallel computers are traditionally divided into two broad subcategories: tightly coupled and loosely coupled systems. In a tightly coupled system at least part of the primary memory is *shared*. All processors have direct access to this shared memory. In a loosely coupled (distributed) system, processors only have access to their own local memories; processors can communicate by sending messages over some kind of communication channel. Tightly coupled systems have the advantage of fast communication through shared memory. Distributed systems, on the other hand, are much easier to build, especially if a large number of processors is required. Initially, programming language and operating system designers strictly followed the above classification, resulting in two parallel programming paradigms: shared variables (for tightly coupled systems) and message passing (for distributed systems).

Meanwhile, the techniques of parallel programming are evolving slowly. Initially, parallel programs were a collection of sequential programs written in traditional sequential languages. These languages were extended with communication instructions to enable message-passing communication. Such synchronizing communication is feasible in problems with completely regular patterns of execution, but it is difficult to control in areas where the execution pattern is strongly data-dependent and irregular.

Therefore, it is clear that changes in programming languages are needed — in particular it should be possible to dynamically create computations, to synchronize them and to allow information exchange between them. Several languages have been proposed with this goal in mind: for instance C.A.R. Hoare proposed an influential model called CSP (Communication Sequential Processes) allowing the definition, activation and synchronization of communicating processes [Hoare, 1985]. However, it is very difficult to build parallel programs with this approach. This is mainly due to the fact that the programmer has to mentally manage several threads of control simultaneously instead of one at the time.

During the past years several high-level languages with mechanisms for parallel programming have been developed to alleviate parallel programming. Even though such high-level languages were designed to express parallelism, they are mostly intended to be implemented and used on ordinary computers with only one central processing unit. Parallel programming is then used as a convenient way of expressing logical relationships between different parts of a complex program, and only secondary for increasing the execution speed.

However, parallel programming is still in its infancy. At present, there is a wide gap between formal approaches to algorithm specification and practical approaches to algorithm implementation. We are developing an approach to parallel algorithm implementation, which exploits the strength of a set-oriented language to program design through rapid prototyping.

Historically, parallelism has been introduced as a means of gaining speed during execution. However, the issue of parallelism as an abstraction mechanism for structured design and programming has gained importance. For instance, in the Parallel Program Design approach of [Chandy and Taylor, 1992] parallelism is used as the main mechanism for achieving program modularity. The focus of the Parallel Program Design approach is the development of programs by the *parallel composition* of simpler components, in such a way that the resulting programs preserve the properties of the components that they compose. Generalizing this approach, in [Carriero and Gelernter, 1992] parallelism is used as a *coordination* mechanism and, accordingly, programming is split in two separate activities: a sequential language is used to build single-threaded computations, whereas a coordination language is used to coordinate the activity of several single-threaded computations. Thus parallel programming consists of *putting together programs* and letting them cooperate.

To summarize, our hypothesis is that

- prototyping of parallel algorithms is a profitable basis for constructing parallel software which meets users' demands and which is amenable to evolving requirements, and that
- generative communication integrated into a set-oriented language is an adequate device to enable rapid prototyping of parallel algorithms.

1.2 A Look at Software Construction

Within the computer science community, it is a well-known fact that the cost to correct an error in a computer system increases dramatically as the system life cycle progresses [Boehm, 1981]. The cost of correcting an error increases by orders of magnitude as the system moves from the development stages of analysis and design, to become most expensive during the maintenance and operation phase. Formal specification and prototyping help to eliminate many of these errors in the very early stages of a project before any production-level code has been written.

In the 1950s, many programs were designed to make optimum use of some specific feature of the hardware. Programs were written to exploit a particular machine-language command or the number of bits in a computer word. Today we know that such optimizations are best left to the last stages of program design, or best left out altogether. Parallel programs today are designed much as the sequential programs were designed in the 1950s: to exploit the message-passing primitives of a language or the network-interconnection structure of an architecture. Experience tells us that it is best to postpone such optimizations until the very end of program development. Issues of performance should be considered, but only at the appropriate time:

> "The basic problem in programming is the management of complexity. We cannot address that problem as long as we lump together concerns about the core problem to be solved, the language in which the program is written, and the hardware on which the program is to execute. Program development should begin by focusing attention on the problem to be solved, postponing considerations of architecture and language constructs." [Chandy and Misra, 1988, page 2]

In order to achieve this goal we should be able to built in the first place an abstract version of a program in a high-level language.

A goal of this thesis is to construct a tool for prototyping parallel algorithms. We shall build this piece of software in a somewhat unconventional way: the informal specification is followed by a formal specification, which serves as the basis for a prototype implementation before the production-level implementation is undertaken. We will motivate this approach to language design and implementation in the following subsections.

1.2.1 Informal and Formal Specifications

Specification of the requirements is the phase in software construction which is concerned with the analysis of the tasks to be performed by the intended software system. Programs, especially parallel programs, are often described informally. Problems such as the dining philosophers problem (see also Sect. 6.5), are posed without using a formal notation. There are advantages to informal descriptions. A problem can be studied without the acquaintance with a formal notation which may turn out to be a burden. Also, pictures such as philosophers sitting around a table exchanging forks are vivid and helpful.

Informal specifications have the advantage that natural-language requirements can be read and approved directly by customers. They can also form the basis of a legal contract. On the other hand, informal requirements are notorious for their incompleteness, inconsistency, and ambiguity. Indeed, these are the problems which first led computer scientists to look for specifications that were formal, and therefore subject to automated checks on their consistency. Mathematical descriptions, however, cannot stand alone. At minimum, prose is needed to relate concepts modeled in mathematics to real-world entities.

The main advantage of natural-language texts is their understandability. One concentrates on this asset rather than trying to use natural language for precision and rigor, qualities for which it is inadequate. Understandability is seriously hindered when natural-language requirements become inadequately long. The length of many requirements documents in actual practice often extending over hundreds or even thousands of pages indicates this misuse of natural language. Natural-language descriptions should remain reasonably short; the exact description of fine points, special cases, precise details, etc., should be left to a formal specification. The advantages of brevity cannot be overemphasized. Formal methods could be a response to the increasing complexity and quality requirements software development industry is confronted with. Some of the better known specification methods such as VDM [Jones, 1990] and Z [Spivey, 1992b] have already been used in industry [Craigen *et al.*, 1993].

An objection that is often voiced against formal specifications relates to the needs of end-users, who request easily understandable documents. Such an objection, we think, is based on an inadequate expectation of what specification is about. There is a need for requirement documents that must be read, checked, and discussed by non-computer scientists, but there is also a need for technical documents used by computer scientists. The difference is very similar to the difference between user requirements and engineering specifications in other engineering disciplines.

Much recent work has been devoted to finding more rigorous mathematical methodologies which — though less accessible — are complete and unambiguous. One methodology is denotational semantics: in this approach each program phrase is given a denotation, or a meaning, as an object of some mathematical domain. It is compositional in the sense that the meaning of each phrase is a function of the meaning of its subphrases. A second methodology is operational semantics: in this approach rules are given for the evaluation of each phrase. Operational semantics is also compositional since the evaluation of each phrase is defined in terms of the evaluation of its subphrases. An operational semantics has the advantage of suggesting a possible implementation, and of easing comparisons with other languages having analogous formal semantics.

Formal specifications can help to expose ambiguities and contradictions, because they force the specifier to describe features of the problem to be solved precisely and rigorously. Formal methods are used to reveal ambiguity, incompleteness, and inconsistency in a system. When used early in the system development process, they can reveal design flaws that otherwise might be discovered only during the costly testing and debugging phases [Wing, 1990]. An interesting result of a formal specification is not only the specification itself but the insight gained about the specified system.

Let us now take a closer look at the definition and implementation of programming languages, since our intention is to design and implement a language for parallel programming. Usually, the syntax (not the semantics) of programming languages is formally described with syntax diagrams or with Backus Naur Form. Therefore, they appear to be good candidates for a formal treatment. To discuss the advantages of formal specifications in the development of a new language, we briefly evaluate the development of the parallel programming language C-Linda.

The definition of C-Linda has been presented informally [Gelernter, 1985] and has included several ambiguities. [Narem, 1989] summarizes four basic types of process creation used in implementations of C-Linda's `eval` operation. These are different interpretations of the informal specification of the `eval` operation. Additional discussions of problems with the semantics of the `eval` operation may also be found in [Leichter, 1989] and in Sect. 7.2 of this thesis. This situation demands a more precise definition. However, informal descriptions are very valuable because it is easy to grasp the gist of their semantics without much effort. The popularity of Linda can in part be ascribed to this property.

To overcome the problems caused by the informal specification, several formalizations of the coordination language Linda have been undertaken. These approaches use structured operational rules such as Plotkin's Structural Operational Semantics (SOS) or Milner's Calculus of Communicating Systems (CCS), Petri Nets, the Chemical Abstract Machine, Term Rewriting Systems, Communicating Horn Clause Logic, Algebraic Specifications, and the formal specification language Z:

SOS, CCS [Hazelhurst, 1990; Jensen, 1990; Callsen *et al.*, 1991; Ciancarini *et al.*, 1992; Jensen, 1994]

Petri Net [Ciancarini *et al.*, 1992]

Chemical Abstract Machine [Ciancarini *et al.*, 1992]

Term Rewriting System [Jagannathan, 1990] (for Scheme-Linda)

Communicating Horn Clause Logic [Bosschere and Wulteputte, 1991] (for Multi-Prolog)

Algebraic Specifications [Anderson *et al.*, 1990] (for TS-Prolog)

Z [Butcher, 1991]

A comparative study of some approaches may be found in [Ciancarini *et al.*, 1992]. To avoid the problems which may be caused by an inconsistent informal specification, we shall present in Chap. 8 the formal semantics of our approach to parallel programming in a set-oriented language by means of the formal specification language Object-Z [Duke *et al.*, 1991]. Our approach combines the set-oriented language ProSet (Chap. 4) with generative communication in Linda (Sect. 3.2.9) to form the parallel programming language ProSet-Linda.

Object-Z is an object-oriented extension to Z. Z is a specification language based on mathematical set theory and logic [Spivey, 1992b]. It has been developed at Oxford University for use in the specification of state-based programs, and has now matured into a valuable and widely used development tool. For example, IBM UK Laboratories at Hursley have used Z to develop new code for the CICS system (the main IBM System/370 on-line transaction processing system). They obtained a significant reduction in the volume of software quality defects for ultimately little additional development cost [Wordsworth, 1991].

The formal specification language Z was chosen as a means for presenting the formal semantics of ProSet-Linda for several reasons. Firstly, Z has many similarities with ProSet: both languages are based on set theory and predicate calculus. This alleviates the access to the formal specification for readers who are familiar with ProSet. Consequently, we capitalize on the similarities when constructing prototypes in ProSet from Z specifications. Furthermore, there are some tools available to support the construction of specifications in Z [Parker, 1991].

A preliminary specification of ProSet-Linda using plain Z has been presented in [Hasselbring, 1992a]. Inferring the operation schemas that may affect a particular state schema requires examining the signatures of all operation schemas in plain Z. In large specifications this is impracticable. This problem and the absence of temporal logic notation in plain Z led us to consider the use of object-oriented extensions of Z to improve our preliminary specification. We shall later use the name Z when both languages, plain Z and Object-Z, are concerned.

There exist several object-oriented extensions of plain Z. [Stepney *et al.*, 1992] is a collection of papers describing various object-oriented approaches for Z — for example Hall's Style, ZERO, the Schuman & Pitt Approach, MooZ, OOZE, ZEST, Z^{++}, and Object-Z — mainly written by the methods' inventors, and all specifying the same two examples. Hall's Style and ZERO provide conventions for writing object-oriented specifications in plain Z. The Schuman & Pitt Approach is more concerned with fundamental issues of composition of schemas and reasoning about the resulting composition than with specifying object-oriented systems, or specifying systems in an object-oriented way.

Object-Z extends Z by introducing a class structure which encapsulates a single state schema with the operations which may affect that state. Object-Z uses the class concept to incorporate a description of an object's state with related operations. Classes, and hence state operations, can be inherited by other classes. The model for a class is based on the idea of a history which captures the sequences of operations and state changes undergone by an instance (object) of the class. One of the advantages of Object-Z is that it enables such constraints to be incorporated directly within the model. Within plain Z, to specify liveness properties, histories must be explicitly defined separately from the state and operation schemas as it has been done in [Hasselbring, 1992a]. Appendix B provides a short introduction to Object-Z.

Z^{++} [Lano and Haughton, 1992] is very similar to Object-Z. Z^{++} also provides history predicates using temporal logic and therefore is an interesting candidate for our purposes. However, Z^{++} does not have a graphical display for schemas and classes. The idiosyncratic syntax is unfamiliar for readers knowing the Z notation. This drawback leads us to the use of Object-Z. MooZ, OOZE and ZEST are other object-oriented extensions of Z, but they do not support history predicates with temporal logic. As the use of history predicates simplifies our specification in Chap. 8 significantly, we actually preferred Object-Z.

Z has no features for specifying parallelism. However, this does not prevent us from specifying a parallel programming language with Z: we shall model concurrency by arbitrary interleaving of atomic transactions which are performed by the acting processes (see also Sect. 8.8). The goal of our formal specification is not specifying as much parallelism as possible, but rather providing a precise specification of the semantics of generative communication in ProSet.

An Object-Z specification consists of a combination of a formal text and a natural language description. The formal text provides the precise specification while the natural language text introduces and explains the formal parts. The formal text has two parts: classes and schemas, which provide a means of structuring the specification, and the mathematical language, which allows for the preciseness of the specification. The mathematical language is based largely on set theory and enables an abstract mathematical view of the objects being specified to be taken. The object-oriented features enable specifications of large systems to be broken into more manageable sections.

The combination of natural language for explanation, and of the classes and schemas for structurization produces specifications that are more readable than only mathematical formulas. In addition, the mathematical nature of the specifications enables implementors to use mathematical proofs to ensure the correspondence of their implementations with the specification.

We do not advocate formal specifications as a *replacement* for natural-language requirements; rather, we view them as a *complement* to natural-language descriptions, and as an aid in improving the quality of natural-language specifications. Formal specifications can serve as a starting point for more precise natural-language requirements. This is so because formal notations lead the specifier to raise questions that might have remained unasked, and thus unanswered, in an informal approach.

Apart from the clear advantage of writing the semantics in a mechanically checkable formalism, a formal specification discloses subtleties as well as difficulties that are otherwise *swept under* the carpet of an imprecise notation. The formal specification emerges as a contract — stating rights and obligations — between language designer and implementor, and it is an abstract, detailed language manual for the programmer.

1.2.2 Formal Specifications and Prototyping

Once we have a formal specification, what can we do with it? Relying on the specification as a basis for the next phases of the software development process is the most obvious use. Formal specifications are needed as an intermediate step between requirement analysis and design.

Traditionally, most large software development projects are conducted along the lines of so-called life-cycle plans [Boehm, 1976]. In life-cycle plans, the principle of proceeding "from the general to the concrete" or "from the problem to the solution" is applied to the organization of a software project. Each activity is viewed as an input-process-output step. Only minimal provision is made for feedback cycles. We refer to [Blum, 1982; Floyd, 1984; Zave, 1984; Doberkat and Fox, 1989; Budde *et al.*, 1992] for critical assessments of the traditional life-cycle approach. Some of the reported problems are:

- Users are excluded from system development.
- Maintenance is unplanned system development.
- Life-cycle plans are unsuitable for project control.

To avoid these problems, we do not try to construct a production-quality program directly from our formal specification. Instead we intentionally build a prototype first (see Chap. 10).

Richard Kemmerer calls this "testing formal specifications" [Kemmerer, 1985]. It is necessary to test specifications early to develop systems which meet their critical requirements and provide the desired functionality. Some of the functional requirements may not be known at design time. In fact, some functional requirements may only arise while evaluating the prototype.

First, a prototype helps the specification writer to debug the specification. It also helps a potential user to experience the capabilities of the system. It is often only through this type of experience that the necessary functional requirements can be discovered. Furthermore, it is better to have the user discover needs early in the production process, and not after the system has been completely implemented and delivered. The prototype provides the user with a vehicle which can be exercised to see if it meets the (sometimes fuzzy) requirements.

Prototyping is used for requirements engineering, risk reduction, specification validation, increased user acceptance, and simplified maintenance of software systems. See Chap. 2 for a more detailed discussion of prototyping.

1.3 Overview

This thesis is organized as follows. Part I presents the setting of our work. This is prototyping (Chap. 2) in connection with parallel programming (Chap. 3). We shall conclude this part with an evaluation of several approaches to high-level parallel programming concerning their suitability for prototyping parallel algorithms in a set-oriented language.

Part II presents our approach to generative communication in set-oriented prototyping. We start in Chap. 4 with a brief introduction to the prototyping language ProSet. Chap. 5 then presents an informal specification for the integration of generative communication into ProSet. We extend this presentation by means of some example programs in Chap. 6, and continue with a discussion of other proposed approaches to extending Linda and some design alternatives to our approach in Chap. 7. The presentation of the formal semantics of generative communication in ProSet by means of the formal specification language Object-Z in Chap. 8 is the glue to Part III, which is concerned with the implementation of ProSet-Linda. We will refine the formal specification into an implementation design in Chap. 9, and continue in Chap. 10 with a prototype implementation in ProSet, which is directly constructed from the formal specification. Chapter 11 discusses some general issues concerning implementing ProSet-Linda. Part IV presents our résumé and an outlook to directions for future research.

The appendices in Part ?? present the abstract grammar for the tuple-space operations in ProSet (Appendix A), the formal specification language Object-Z (Appendix B), and the types of all names defined globally in the formal specification of Chap. 8 and 9 (Appendix C). The index of formal definitions provides a summary of the names defined in the formal specification. There is also an index of explained Object-Z symbols and keywords. The general index and the bibliography can be found at the end of this document.

1.4 Acknowledgments

The discussions on various aspects of this work with Henri Bal, Nick Carriero, Mike Factor, Michael Goedicke, Jerry Leichter, Greg Wilson, and many others were very helpful. Furthermore, the discussions on formal semantics with Stephen Gilmore and Keld Kondrup Jensen were very helpful to improve our preliminary formal specifications. Last but not least, the encouragement and continuous support by Ernst-Erich Doberkat is gratefully acknowledged.

Part I

Setting

Chapter 2

Prototyping

One of the more recent approaches for complementing the traditional model of software production using the life cycle approach is rapid prototyping. Prototyping refers to the well defined phase in the production process of software in which a model is constructed which has all the essential properties of the final product, and which is taken into account when properties have to be checked, and when the further steps in the development have to be determined [Floyd, 1984]. In this chapter, we will sketch goals and approaches to prototyping. Chapter 4 will later present a short introduction to the basic concepts of the prototyping language ProSet. We want to note that for us a prototype is a model, and that this model taken as a program has to be executable so that at least part of the functionality of the desired end product may be demonstrated on a computer. Therefore, prototyping involves producing early working versions (prototypes) of the future application system and experimenting with them.

It is important to note that the material of this chapter is *not* the original work of the author of this thesis. This chapter is essentially compiled from the standard literature to prototyping [Dearnley and Mayhew, 1983; Budde *et al.*, 1984; Hekmatpour and Ince, 1988; Doberkat and Fox, 1989; Budde *et al.*, 1992]. We intend to provide a short introduction to goals and approaches to prototyping, and a guide to further reading. To make fluent reading easier we do not indicate the source for each argument.

2.1 Goals of Prototyping

Software prototypes are used somewhat differently from hardware prototypes. For the most part, hardware prototypes are used to measure and evaluate aspects of proposed designs that are difficult to determine analytically. For example, simulation is widely used to estimate throughput and device utilization in proposed hardware architectures. Although software prototypes can be used likewise to determine time and memory requirements, they usually focus on evaluating the accuracy of problem formulations, exploring the range of possible solutions, and determine the required interactions between the proposed system and its environment. The nature of a software prototype is also different from, for example, an architectural model: a software prototype actually demonstrates features of the target system in practical use, and is not merely a simulation of them. We will use the term *prototyping* as a synonym for software prototyping in the remainder of this thesis. Typical properties of prototypes are:

- Prototypes are not intended to be the final system.
- Prototypes are not necessarily representative of a complete system. They are miniature models of proposed systems.
- Prototypes are functional after a reasonable amount of effort.

- Prototypes are flexible, i.e., modifications require minimal effort.
- Prototypes are an executable specification for the developer.
- Prototypes are a model used to determine design appropriateness.
- Prototypes are a feasibility study.
- Prototypes are a means for providing users of a proposed application with a functional representation of key parts of the system before system implementation.
- Prototypes are a basis for discussion and an aid to decision-making. They are something that promotes communication between developers and users.

Typical applications of prototyping are:

Requirements engineering Prototyping improves communication through demonstration. This enables an earlier and more effective dialogue between users and developers, helps to expose unstated assumptions, and triggers some of the inevitable requirements changes early in the process of software engineering. Consequently, prototyping reduces rebuilding.

The traditional approach to requirements engineering is to interview potential users and prepare a requirements document, which is reviewed and modified until there are no more objections. This approach has not worked well in practice, because users have not been effective in discovering requirements errors when they review the document, which often leaves room for misunderstandings.

A common problem with software products is that the users of the system may not be fully aware of what they require and they may be unable to communicate their desires to the development team. Using a prototype, the user can interact with the system and can discover requirements deficiencies early, enabling rapid correction of the requirements.

Risk reduction Prototyping reduces risks by making communication between users and developers more specific, by helping to determine a proposed design's unknown properties, and by providing a basis for assessing the feasibility and performance of alternate designs. A prototype can be used as a feasibility study.

Specification validation Prototyping is a way to validate specifications. Validation attempts to ensure that all parties — users and developers — interpret the specification in the same way. Without this assurance, developers run a high risk of developing and testing software built on inadequate specifications. Prototyping should be integrated with the formulation and recording of specifications and assessment of a design's feasibility. The opportunity for inadequate functional specifications is greatly reduced with prototyping.

User acceptance With the traditional life cycle, users continue to complain that systems are late, contain errors, and are often not what they wanted and thought they asked for. This is like *talking to the stars*: at the time when the answer arrives, the question has been forgotten [Blum, 1982]. With prototyping, a *what you see is what you get* approach eliminates the bulk of the communication problems. It allows delivery of systems as quickly as possible and permits the correction of errors due to bad specifications in early relatively inexpensive phases.

When users are forced to participate, they can gain confidence in a system. They *see* first-hand the problems and errors, and they also *see* the mistakes getting resolved quickly before the mistakes haunt the system for life. When users *command* the system directly, there is little chance for communication problems to arise between user and developer. Prototypes can also serve to train users of future systems.

Maintenance Prototyping can also provide many of the same benefits when a system's requirements change after delivery. Especially when the proposed changes are so drastic that they fundamentally alter a system's goals, the production program may not be a good basis for evaluating them. In these cases, prototyping can reduce uncertainty and the number of times the production program must be changed before a satisfactory result is obtained.

It is better to base maintenance changes on the flexible prototype instead on the optimized implementation, because optimizing transformations often introduce conceptual dependencies that increase the fraction of the code affected by a change in the requirements.

Prototyping has been developed as an answer to deficiencies in the traditional life cycle model, but it should not be considered as an alternative to this model. It is rather optimally useful when it complements the life cycle model. It is plausible that prototyping may be used during the early phases of the design. For a discussion of the relation between prototyping and the life cycle model we refer to [Dearnley and Mayhew, 1983; Doberkat and Fox, 1989; Budde *et al.*, 1992].

2.2 Approaches to Prototyping

The idea of prototyping is being adopted in software engineering for different purposes: prototypes are used exploratively to arrive at a feasible specification, experimentally to check different approaches, and evolutionary to build a system incrementally. Prototypes may be classified as follows (see also [Dearnley and Mayhew, 1983; Floyd, 1984; Doberkat and Fox, 1989; Budde *et al.*, 1992]):

Throwaway A throwaway prototype describes a product designed to be used only to help the customer identify requirements for a new system. All elements of the working prototype will be discarded, as intended, after system requirements have been identified. Only the derived requirements will be maintained, paving the way for work on the real system. Such exploratory prototyping is used where the problem at hand is unclear. Initial ideas are used as a basis for clarifying user and management requirements with respect to the future system.

Throwaway prototypes are sometimes perceived as a waste of effort. The reason is that developing code that will be thrown away is considered a waste of resources. It is true that prototype code is often too inefficient and insufficiently general to be directly incorporated into a final product. But this argument ignores the fact that production-quality code often must be discarded, because it is based on incorrect requirements. It is most cost-effective to correct the requirements by evaluating and discarding a relatively inexpensive prototype instead of an expensive product.

The main contribution of throwaway prototypes is not code, but the insight they give analysts into correct system behavior and the structure of a feasible design.

However, the reliance of throwaway prototypes signals insufficient technological support for recording, transforming, and implementing specifications and designs.

Experimental This form of prototyping focuses on the technical implementation of a development goal. By means of experimentation, users are able to further clarify their ideas about the type of computer support required. The developers, for their part, are provided with a basis for appraisal of the suitability and feasibility of a particular application system. An essential aspect here is the communication between users and developers on technical problems and questions relating to software features, which cannot be clarified analytically.

Evolutionary In evolutionary prototyping, a series of prototypes is produced that converges to an acceptable behavior, according to the users' feedback from prototype

> demonstrations. Parts of the description and design of each version are refined in the next version to the extent that the two versions share common requirements, subfunctions, and data. Once the series has converged, the result may be turned into a software product by transformations.
>
> Here, prototyping is not merely used as a device in the context of a single development project. Evolutionary prototyping is rather a continuous process for adapting the model of an application system to changing organizational constraints.

Note that exploratory prototyping is not restricted to the development of throwaway prototypes. Another classification of prototyping is the subdivision into horizontal and vertical prototyping. In horizontal prototyping, only specific layers of the system are built, e.g., the graphical user interface layer or functional core layers. In vertical prototyping, a selected part of the target system is implemented completely. This technique is appropriate where the system's functionality and implementation options are not sufficiently defined yet.

The approach to prototyping with ProSet is an evolutionary development in versions. This approach is reasonable, because the developed product may manipulate the environment in which it is used. Introducing a new system often triggers organizational restructuring, changing how the organization does its business. This restructuring may change users' responsibilities and procedures, thus triggering changes in the evolving system. Therefore, the system itself becomes a driving force for changes in its requirements. Thus the prototype has to evolve in accordance with the changing environment. The ordering of development steps in the traditional life cycle model is mapped here into successive development cycles. Users are involved in the system development process which supports the communication between users and developers.

Prototypes should be built in high-level languages to make them rapidly available. To be useful, prototypes must be built rapidly, and designed in such a way that they can be modified rapidly. Consequently, a prototype is usually not a very efficient program since the language should offer constructs which are semantically on a very high level, and the runtime system has a heavy burden for executing these highly expressive constructs. To obtain a more efficient production level version program transformations are desirable to refine the prototype design into a production-quality product [CIP, 1987; Partsch, 1990]. ProSet contains a Pascal-like subset that facilitates evolutionary prototyping by allowing a program to be refined into successively finer detail while staying within the language.

Chapter 4 will present a brief introduction to the prototyping language ProSet. For comparative discussions of high-level languages such as Lisp, PROLOG or SETL, which may be used to develop prototypes, we refer to [Doberkat and Fox, 1989; Budde *et al.*, 1992].

2.3 Summary

This chapter presents a short introduction to goals and approaches to prototyping, and a guide to further reading. Prototyping constitutes the first part of the setting for this thesis.

Chapter 3

Parallel Programming

There has been particular attention on parallel programming and processing within the computer science community in the last years. In this chapter, we will sketch goals and approaches to parallel programming and processing in general and approaches to high-level parallel programming models and languages in particular. We shall conclude with an evaluation of these approaches to high-level parallel programming concerning their suitability for prototyping parallel algorithms in a set-oriented language.

3.1 Goals of Parallel Programming

There are two schools of thought regarding the proper approach to parallel programming and processing: the "program in sequential languages and let the compiler extract the parallelism" school and the "provide constructs for expression of parallelism" school.

The first school sidesteps the issue altogether. Instead of proposing new software models, it relies on the automatic generation of parallel versions of ordinary sequential programs [Banerjee *et al.*, 1993]. This is the detection of implicit parallelism. Driven primarily by the colossal investment of time and money represented by existing programs much effort has been invested for developing compilers that automatically transform sequential programs into parallel machine code. A parallelizing compiler finds the parallelism hidden in a sequential program by analyzing its structure for operations that in fact can be done simultaneously, even though the program specifies they be done one by one. It then generates code that reflects the implicit parallelism it has found.

A parallelizing compiler can often tell by inspection whether loop iterations really must be done sequentially, as the program prescribes. If each new calculation depends on values calculated in a previous iteration, sequentiality is required. Otherwise the compiler might well be able to convert the loop into parallel machine code, allowing the computer to do all calculations simultaneously.

Data-flow computers [Srini, 1986] offer another way to take advantage of implicit parallelism. These machines are usually aimed at functional programming languages and data-flow languages [Ackermann, 1982]. Functional languages make it easier for the machine to decompose the program into parallel activities, since side effects are not allowed. Languages for data-flow computers often allow assignments, but the value of a variable can be set once only. After a variable's value has been set, other parts of the program can use the variable freely and simultaneously, because its value cannot change. Implicit parallelism is also present in logic programming languages (see Sect. 3.2.4).

If parallelism can be found automatically, why to create new, explicitly parallel software models? In fact automatic transformations work well only for certain kinds of programs. A parallelizing compiler, for example, transforms a sequential program in parallel machine code on the basis of the program's expected behavior. Many programs — numerical computations in particular — do unfold in a predictable way, and so a compiler can reliably determine which parts of the task can be done in parallel. However, most programs — symbolic computations such as discrete optimization problems [Grama and Kumar, 1992] in particular — have execution patterns that are complicated or unpredictable, and here the implicit parallelism is harder to discover.

Parallelizing compilers are often used because of the high investment in existing application software. Similar to digitizing old recordings, which cannot produce the quality of new digital recordings, parallelizing sequential algorithms does not work as well as algorithms genuinely conceived for parallel implementations. Automatic parallelization can discover the parallelism latent in an existing algorithm, but it cannot invent new parallel algorithms. Programmers who can express their ideas in a parallel way sometimes invent entirely new ways of solving problems. In order to embody their inventions in working programs they need languages that allow parallelism to be expressed explicitly — languages based on parallel software models.

Opportunities for automatic detection of parallelism in existing programs are, therefore, limited and furthermore, in many cases the formulation of a parallel program is more natural and appropriate than a sequential one. Programming explicitly parallel programs is not much harder than programming sequential programs, if the problem lends itself to a parallel solution. Consequently, we argue that the best way to get a parallel program is to write a parallel program! Hence, there does not remain the question if we should provide explicit parallelism to the programmer at all. The remaining question is: How to provide explicit parallelism? We shall address this subject in Sect. 3.2.

There exist several motivations for programming parallel applications:

1. Decreasing the execution time for an application program.
2. Increasing the fault-tolerance.
3. Exploiting inherent parallelism of the application.

Achieving speedup through parallelism is a common motivation for executing an application program on a parallel computer system. Another motivation is achieving fault-tolerance: for critical applications like controlling a nuclear power plant, a single processor may not be reliable enough. Parallel computing systems are potentially more reliable: as the processors are autonomous, a failure in one processor does not affect the correct functioning of the other processors. Fault-tolerance can, therefore, be increased by replicating functions or data of the application on several processors. If some of the processors crash, the others can continue the job. We will return the this subject in Sect. 7.14.

The main motivation for integrating explicit parallelism into a prototyping language is to provide means for modeling inherently parallel applications. Consider, for instance, distributed systems such as air-traffic-control and airline-reservation applications, which must respond to many external stimuli and which are therefore inherently parallel. To deal with nondeterminism and to reduce their complexity, such applications are usually structured as independent parallel processes. Similarly, a company with multiple offices and factories may need a computing system which enables people and machines at different sites to communicate with each other. Such a system has to run on distributed hardware and, thus has to be programmed in a parallel way.

Consequently, a prototyping language emphasizes *logical parallelism* rather than *physical parallelism*. We quote for a definition of logical parallelism and physical parallelism:

> "Physical parallelism is related to the implementation; it corresponds to the distribution of tasks on several processors. By logical parallelism, we mean the possibility of describing a program as a composition of several independent tasks. Of course, a particular implementation can turn logical parallelism into physical parallelism, but these two notions have very different natures: the former is a program-structuring tool, whereas the latter is an implementation technique." [Banâtre and Métayer, 1993, page 98]

Therefore, logical parallelism is the *potential* for physical parallelism. Sometimes the term *concurrency* is used for logical parallelism [Ben-Ari, 1990]. As we are mainly interested in logical parallelism, we shall use the terms *concurrency* and *parallelism* as synonyms for logical parallelism in this thesis.

3.2 Approaches to Parallel Programming

There are two issues which must be addressed in designing a language for parallel programming:

- How to achieve parallel execution?
- How to coordinate parallel executing parts of a program?

The first issue is concerned with process creation and termination, and the second issue is concerned with communication and synchronization between cooperating parallel processes. An issue related to synchronization is *nondeterminism*. A process may want to wait for information from any of a group of other processes, rather than from one specific process. Since it is not known in advance which member (or members) of the group will provide its information first, such behavior appears to be nondeterministic for the waiting process. To program such behavior, a notation is needed to express and control nondeterminism. As we shall see, each of these issues may be addressed to some degree in a given language, and they may be resolved in quite different ways.

Many languages for parallel programming have evolved during the last years, making the choice of the most suitable language for prototyping parallel algorithms a difficult one. More important the underlying *models* of the languages differ widely. Below, we shall take a closer look at several high-level parallel programming models and languages which may be suitable for prototyping parallel algorithms and applications. We conclude with an evaluation in Sect. 3.2.10. The models we consider for presentation and evaluation are:

1. Data parallelism (Sect. 3.2.1)
2. Parallel object-oriented programming (Sect. 3.2.2)
3. Parallel functional programming (Sect. 3.2.3)
4. Parallel logic programming (Sect. 3.2.4)
5. Unbounded nondeterministic iterative transformations (Sect. 3.2.5)
6. Multiset transformations (Sect. 3.2.6)
7. Virtual shared memory (Sect. 3.2.7)
8. The shared data-object model (Sect. 3.2.8)
9. Generative communication (Sect. 3.2.9)

As we shall see, mainly models which emphasize some kind of *shared data* were considered. Our presentation excludes the basic model of *message passing*. This basic model is that of a group of sequential processes running in parallel and communicating through passing messages. This model directly reflects the distributed memory architecture, consisting of processors connected through a communications network. Many variations of message passing have been proposed. With *asynchronous* message passing, the sender continues immediately after sending the message. With *synchronous* message passing, the sender must wait until the receiver accepts the message. Remote procedure call and rendezvous are two-way interactions between two processes. Broadcast and multicast are interactions between one sender and many receivers. Languages based on the message passing model include CSP, occam, Ada, SR, and many others. As these languages with their variations of message passing have been studied extensively in the literature, we refer to [Bal *et al.*, 1989] for an overview, and do not discuss them in detail here. However, we shall include this model in our evaluation in Sect. 3.2.10. Note, however, that the parallel object-oriented languages, which shall be discussed in Sect. 3.2.2, rely on message passing as their device for communication.

As the prototyping language ProSet (Chap. 4), which is a successor to SETL [Schwartz *et al.*, 1986], is an essential basis for Part II of this thesis, we mention two message-passing variations of SETL.

PARIS (for PARallel Interpretation of Setl) extends SETL with simple primitives for sending and receiving messages [Korneev *et al.*, 1991]. PAISLey is a set-based functional programming language whose set-based aspects are similar to SETL [Zave and Schell, 1986]. Parallelism in PAISLey is based on a model of event sequences and used to specify functional and timing behavioral constraints for asynchronous parallel processes. The processes communicate via *exchange functions*. An *exchange function* consists of a pair of function calls, one in each of two different processes. During execution, when the two processes both reach the evaluation of their half of the exchange function, the argument of one function is passed as the return value to the other and vise versa (i.e., they exchange their arguments). The connection between the two functions is established via *channel attributes* of the function calls. Exchange functions may be compared with Ada's rendezvous concept [Bal *et al.*, 1989].

There exist some approaches to develop prototypes for parallel programs on the basis of Petri-nets [Reisig, 1985] or data-flow diagrams [DeMarco, 1978]. Petri-Nets and data-flow diagrams can be regarded as graphical representations of message-passing systems. Therefore, they are often used to build prototypes for message-passing programs. For instance, prototypes for occam programs are developed in [Bréant and Pavoit-Adet, 1992] with Petri-nets and in [Jones *et al.*, 1990] with data-flow diagrams. In [Ma and Hintz, 1992], Petri-nets in combination with data-flow diagrams are used to develop prototypes for occam programs. Extended data-flow diagrams are used in [Levy and Pavlides, 1990] to support prototyping of distributed systems. The language SEGRAS [Krämer, 1991] is based on an integration of algebraic specifications and high-level Petri-nets. Data objects are specified as abstract data types, while dynamic behavior is specified graphically by means of high-level Petri-nets. A subset of the language is executable to support specification-level prototyping. Execution is based on the operational semantics of the language combining term rewriting and Petri-net simulation.

For some applications, the basic model of message passing may be just what is needed. This is, for example, the case for an electronic mail system. For other applications, however, this basic model may be too low-level and inflexible. In particular, the lack of a global name space forces algorithms to be specified at a relatively low level, since it is complicated to simulate shared memory. This greatly increases the complexity of programs, and also restricts algorithm design choices, inhibiting experimentation with alternate algorithm choices or problem decompositions. Therefore, several alternative models have been designed for parallel programming, which provide higher-level abstractions. These languages emphasize some kind of shared data.

The traditional method for communication and synchronization with shared data is through shared variables. The use of shared variables for coordination of parallel processes with, e.g., semaphores or critical sections has been studied extensively. We assume a familiarity with this material and refer the uninitiated reader to [Andrews and Schneider, 1983] or [Andrews, 1991]. For instance, Proteus [Mills *et al.*, 1991] is a SETL-like language where the communication between parallel processes is through shared variables. We regard shared variables as a low-level medium for coordination, because the synchronization, which is necessary to prevent multiple processes from simultaneously changing the same variable (avoiding lost updates), is difficult. Several other coordination models based on shared data exist, however, which are better suited for parallel programming.

A well developed coordination device is the distributed data structure. Distributed data structures are data structures that can be manipulated simultaneously by several processes. A distributed data structure is, therefore, a shared data object to which many processes may append and remove information. Processes communicating via distributed data structures do so with minimal coordination and book-keeping: processes may deposit data without being aware of the receivers who will access it. Processes may access data without being aware of the producers who generated it. This implies asynchronous behavior, since the generation of information is decoupled from its consumption.

Distributed data structures are found in a number of languages. Distributed data structures in form of arrays provide a shared memory model (Sect. 3.2.1). Concurrent object-oriented languages that permit an object to receive messages simultaneously from several processes also support a form of distributed data structure, since the state of such an object may be visible to many processes concurrently (Sect. 3.2.2). Multilisp's *futures* (Sect. 3.2.3) support distributed data structures since a complex structure, whose elements are futures, may be examined and manipulated by many processes simultaneously. The write-once shared logical variables (Sect. 3.2.4) can also be viewed as distributed data structures: an attempt to access a shared logical variable blocks until the shared logical variable is set. The Unity model (Sect. 3.2.5) provides distributed data structures in form of sets of mutable variables, and the Gamma model (Sect. 3.2.6) provides distributed data structures in form of multisets. Distributed data structures can also be built in virtual shared memory (Sect. 3.2.7). Orca (Sect. 3.2.8) supports distributed data structures through the use of shared data-objects.

However, the most well-developed use of distributed data structures is found in the Linda programming model of generative communication. The fundamental mechanism by which distributed data structures are built and manipulated in Linda is the *tuple space*, a data abstraction which resembles a shared associative memory. We shall discuss generative communication in detail in Sect. 3.2.9.

3.2.1 Data Parallelism

Data parallelism extends conventional programming languages so that some operations can be performed simultaneously on many pieces of data. All the elements in a list or in an array can be updated at the same time, for example, or all items in a data base are scanned simultaneously to see if they match some criterion. Data-parallel operations appear to be done *simultaneously* on all affected data elements. This kind of parallelism is opposed to *control parallelism* that is achieved through multiple threads of control, operating independently. According to Flynn's taxonomy of computer architectures, the data-parallel programming model is based on the *single-instruction-stream/multiple-data-stream* (SIMD) model as opposed to the *multiple-instruction-stream/multiple-data-stream* (MIMD) model. The SIMD programming model is *synchronous* because all active processing elements execute the same operation simultaneously.

Several proposals involve annotating sequential imperative languages (e.g. FORTRAN) with directives for mapping processes and data to processors. An example in Kali [Mehrotra and Rosendale, 1991]:

```
processors procs(p)
real A(n), B(n), C(n)
dist A(block), B(cyclic), C(block_cyclic(b))
```

Given a processor array, the programmer must specify the distribution of data structures across the processors. Here, the arrays A, B, and C are distributed across the p processors of the one-dimensional processor array *procs*. Array A is distributed by blocks such that each processor receives a contiguous block of elements of the array. Conversely, array B has its rows cyclically distributed. Array C is distributed in a block-cyclic fashion with size b. That is, the elements of C are first divided into blocks of size b, and then these blocks are cyclically distributed across the set of processors. Parallel operations on these distributed data structures are then specified by **doall** loops:

```
      doall 10 i=1, n on owner(A(i))
         A(i) = ...
         .
         .
         .
10    continue
```

The iterations of a **doall** loop must not have inter-iteration dependencies. That is, any memory location assigned to in one iteration cannot be accessed or modified in any other iteration. This allows the iterations to be executed in parallel. There is an implied synchronization at the beginning and at the end of the parallel loop: all threads start concurrently and have to finish execution of their iterations before any statement following the loop is executed. The **on** clause specifies the processor on which each loop invocation is to be executed. The system-defined function **owner** yields the home processor of its argument. Such programs are usually executed on vector processors and processor arrays [Duncan, 1990].

An approach to data-parallel image processing can be found in Paragon [Reeves, 1991] which provides data-parallel operations on so-called *Parrays* (parallel arrays). Data-parallel operations in Paragon are applied as function calls on the Parrays, and not in parallel loops to avoid overspecification of loops over such parallel arrays. Another approach to data-parallel image processing which is based on the computational model of cellular automata can be found in CELIP [Hasselbring, 1990], where the data-parallel operations are applied through overloading predefined operators. These approaches avoid the explicit use of parallel loops, because parallel loops have fundamentally different semantics compared to sequential loops.

Since ProSet is a successor to SETL, we take a closer look at a data-parallel extension of SETL for a comparison with our work. In Parallel SETL [Hummel *et al.*, 1991], parallelism is introduced into SETL through the use of parallel iterators, which are used in explicit and implicit loops over sets and tuples. For example, the instructions in the following loop are executed in parallel and not executed in sequential iterations:

```
for i par_over [1..1000] do
     statements
end do;
```

There is no explicit tasking like futures in Multilisp (Sect. 3.2.3) or `eval` in Linda (Sect. 3.2.9) in Parallel SETL. If parallel loop iterations in Parallel SETL update the same global variable, then one wins, and the others are lost. In Parallel SETL, multiple writes are *not* considered harmful and enable chaotic algorithms to be expressed. This may cause the well known problems with lost updates of shared variables, because access to shared variables is not synchronized:

> "After code written in Parallel SETL has been verified on a uniprocessor, there is no guarantee that the algorithm and code will work on a multiprocessor. The ability to execute the code on a uniprocessor provides one level of testing and analysis, but will not exercise the synchronization facilities, nor permit the discovery of race conditions or other potential problems with parallel code." [Hummel *et al.*, 1991]

In our approach to parallel programming, which will be presented in Part II, the effects of accessing shared data are well defined and thus our programs shall run with the same functionality on a uniprocessor as they do on a multiprocessor (but probably not with the same speed). Note, however, that in Parallel ISETL [Jozwiak, 1993], which is another data-parallel extension of SETL, access to shared variables is synchronized by the explicit use of semaphores to avoid lost updates and, therefore, chaotic algorithms.

Bibliographic Notes

For an account to data parallel algorithms see [Hillis and Steele, 1986] and for an account to data-parallel programming see [Quinn and Hatcher, 1990]. Flynn's taxonomy of computer architectures is published in [Flynn, 1966]. The example in Kali is from [Mehrotra and Rosendale, 1991].

3.2.2 Parallel Object-Oriented Programming

An approach to imperative programming which has gained widespread popularity is that of object-oriented programming. In this approach, an object is used to integrate both data and the means of manipulating that data. Objects interact exclusively through message passing and the data contained in an object is visible only within this object itself. Objects are used for structuring programs.

As already noted, we do not present the classical model of messages passing in detail. Instead we take a closer look at parallel object-oriented languages where communication between objects is always accomplished through message passing because sharing data among objects violates the encapsulation principle.

The behavior of an object is defined by its class, which comprises a list of operations that can be invoked by sending a message to an object. All objects must belong to a class. Objects in a class have the same properties and can be manipulated using similar operations. The definition of an object class can act as a template for creating instances of the class. Each instance has a unique identity, but has the same set of data properties and the same set of operations which can be applied to it.

Inheritance allows a class to be defined as an extension of another (previously defined) class. Typically when a new class is created, a place for it is defined within the class hierarchy. The effect of this is that the new class inherits the state and operations of its superclass in the hierarchy. Objects may inherit features from more than one class in some approaches (multiple inheritance).

There are several possibilities for the introduction of parallelism into object-oriented languages, viz.:

- Objects are active without having received a message.
- Objects continue to execute after returning results.
- Messages are sent to several objects at the same time.
- Senders proceed in parallel with receivers.

These possibilities can be realized by associating a process with each object. Just as a parallel-processing environment implies multiple processes, a parallel object-oriented system spawns multiple objects, each of which can start a thread of execution. Objects and processes, however, are independent of each other. Processes invoke operations in objects. Objects are usually addressed by an object *reference* (returned upon creation of the object) or by a global object name.

Parallel object-oriented languages use three types of communication: synchronous, asynchronous, and eager invocation. Synchronous communication uses remote procedure calls. It is easiest to implement, but sometimes wastes time because of the requirement for both the sender and receiver to *rendezvous*. Asynchronous communication eliminates the wait for synchronization and can increase concurrent activity. Eager invocation, or the *futures* method, is a variation of asynchronous communication (see also our discussion of futures in Multilisp in Sect. 3.2.3). As in asynchronous operation, the sender continues executing, but a *future variable* holds a place for the result. The sender processes until it tries to access the future variable. If the results have been returned, the sender continues; if not, it blocks and waits for the results.

In object-oriented programming, communication and computation are subsumed by objects and their operations (also called methods). The model is inherently distributed, since it emphasizes distinct, independent objects communicating via messages rather than centralized control constructs. Facilities for synchronization and mutual exclusion are usually supplied in form of semaphores and monitors [Andrews and Schneider, 1983]. Sloop [Lucco, 1987] supports *indivisible* objects, for which only one operation invocation at a time is allowed to execute.

Inheritance complicates synchronization. When a subclass inherits from a base class, programs must sometimes redefine the synchronization constraints of the inherited method. If a single centralized class explicitly controls message reception, all subclasses must rewrite this part each time a new operation is added to the class. The subclass cannot simply inherit the synchronization code, because the highest-level class cannot invoke the new operation.

Bibliographic Notes

The book [Yonezawa and Tokoro, 1987] is a collection of papers on various parallel object-oriented languages, and [Wyatt *et al.*, 1992] is a more recent survey of parallel object-oriented languages. We refer to these publications for an overview about parallel object-oriented languages.

Concurrent Smalltalk [Horwat *et al.*, 1989], for example, defines *all* messages as asynchronous, and synchronization for their return values is provided by the explained future method. Several methods may execute concurrently on an object, but synchronization across tasks is managed by the programmer using locks and semaphores. Concurrent Smalltalk introduces the concept of a *distributed object* with a single name but a distributed state.

Parallelism is achieved in Emerald [Jul *et al.*, 1988] by making objects active. Emerald provides a shared name space for objects, together with a location-transparent invocation mechanism. It is unusual in that it supports task and object migration.

The actor model [Agha, 1986] is not object-oriented as is does not support any form of inheritance. It is more appropriately considered to be *object-based*. An actor is simply an object which responds to messages, but it can only respond to a single message at a time. Parallelism is supported in many ways. Messages can be sent to several actors, so that each is responding to its own message. An actor responds to exactly one message, then *dies*. One of the things that it must do before it dies is to specify a replacement which will handle any additional messages sent to that actor. A message queue is associated with each actor to hold incoming messages in the order of arrival.

3.2.3 Parallel Functional Programming

A functional program comprises a set of equations describing functions and data structures which a user wishes to compute. The application of a function to its arguments is the only control structure in pure functional languages. Functions are regarded in the mathematical sense in that they do not allow side effects. As a consequence a value of a function is determined solely by the values of its arguments, a property which is referred to as *referential transparency*. The absence of side effects accounts for the well-known *Church-Rosser Property*, which essentially states that no matter what order of computation is chosen in evaluating an expression list, the program is guaranteed to give the same result (assuming termination).

Therefore, functional programs are inherently parallel. Because they are free of side effects, each function invocation can evaluate all of its arguments and possibly the function body in parallel. The only delay may occur when a function must wait on a result being produced by another function. The real problem is not discovering parallelism but reducing it so as to keep the overhead on an acceptable level. Parallel functional languages address this problem by allowing the programmer to insert annotations which specify when to create new threads of control.

Multilisp is such a parallel functional language [Halstead, 1985]. Multilisp augments Scheme with the notion of *futures* where the programmer needs no knowledge about the underlying process model, inter-process communication or synchronization to express parallelism. He only indicates that he does not need the result of a computation immediately (but only in the "future") and the rest is done by the runtime system. Instead of returning the result of the computation, a placeholder is returned as result of process spawning. The value for this placeholder is undefined until the computation has finished. Afterwards the value is set to the result of the parallel computation: the future *resolves* to the value. Any process that needs to know a future's value will be suspended until the future resolves thus allowing concurrency between the *computation* of a value and the *use* of that value. The programmer is responsible for ensuring that potentially concurrently executing processes in Multilisp do not affect each other via side effects. An example:

```
(let ((x (future expr1))
      (y expr2))
     ( body ))
```

The value for `x`, which will be the result value of *expr1*, is evaluated concurrently to *expr2* and *body*. The value for `y`, which will be the result value of *expr2*, is evaluated before the evaluation of *body* will be started. When *body* needs the value of `x`, and `x` is not yet resolved, it *touches* the future of `x` and is suspended until the future resolves. Most operations, for example arithmetic, comparison, type checking, etc., touch their operands. This is opposed to simple *transmission* of a value from one place to another which does *not* touch the value, for example by assignment, passing as a parameter to a procedure, returning as a result from a procedure, and incorporating the value into a data structure. Transmission can be done without waiting for the value. The semantics of futures is based on *lazy evaluation*, which means that an expression is not evaluated until its result is needed.

Bibliographic Notes

Collections of papers on several parallel functional languages may be found in [Ito and Halstead, 1990; Szymanski, 1991]. [Schreiner, 1993] provides an annotated bibliography on parallel functional programming. The *Church-Rosser Property* is proven in [Barendregt, 1985] for the λ-calculus.

There exist approaches to futures very similar to Multilisp's futures, for instance, in Qlisp [Gabriel and McCarthy, 1988]. The object-oriented parallel language COOL [Chandra *et al.*, 1990] adds futures to C++. Concurrent Smalltalk [Horwat *et al.*, 1989] adds futures to Smalltalk (see also Sect. 3.2.2).

So-called *para-functional* programming in ParAlfl [Hudak, 1986] is another approach to parallel functional programming. ParAlfl is a functional language which provides mechanisms for mapping a program onto specific multiprocessor topologies. The mapping is accomplished by annotating expressions so as to indicate the processor on which they will be executed. An expression (**future** *expr*) in Multilisp is written in ParAlfl as (*expr* $on *proc*), which declares that *expr* is to be computed on the processor identified by *proc*. The expression *proc* must evaluate to a processor identification. It is assumed in ParAlfl that processor identifications are integers and that there is some predefined mapping from those integers to the physical processors they denote.

3.2.4 Parallel Logic Programming

Logic programming languages, of which PROLOG is best known, express programs as a set of *clauses*, which may be read procedurally or declaratively. For instance, the following clauses:

```
A :- B,C,D
A :- E,F
```

can be interpreted as "to do A, do either B, C, and D, or do E and F". Alternatively, we can view it as "A is true, if either B, C, and D are true, or E and F are true".

PROLOG programs express two distinct forms of parallelism. Firstly, several different clauses may be evaluated separately. This is called OR-parallelism, since only one of them must succeed. Secondly, each subgoal (in the above example B, C, D, E, and F are subgoals) can be executed in parallel, although data dependencies may limit the extent of parallelism. This is called AND-parallelism, since all of the subgoals must succeed for the clause to succeed. AND-parallelism is the simultaneous reduction of several different subgoals in a goal; OR-parallelism is the simultaneous evaluation of several clauses for the same goal. AND/OR-parallelism is implicit parallelism similar to the implicit parallelism found in functional languages, which does not give additional expressiveness.

Similar to functional programs, for logic programs the real problem is not discovering parallelism but reducing it so as to keep the overhead on an acceptable level. In the Aurora approach [Lusk *et al.*, 1988] PROLOG has been extended with the predicate `parallel`, which defines a rule explicitly as OR-parallel executable. This approach is similar to Multilisp's annotations which specify when to create new threads of control (see Sect. 3.2.3).

AND-parallel committed choice logic is a new approach to parallel logic languages which uses guards in the clauses [Shapiro, 1989]. Subgoals are unified in parallel. This approach uses shared logical variables as a communication medium. The difference to PROLOG is that the clauses in the program must be written in a way that avoids backtracking, as this is not allowed in committed choice logic. In OR-parallel committed choice logic all clauses that match a goal are tried in parallel and when one can be unified, the execution commits to this clause. If more clauses can be unified, one is chosen nondeterministically. The committed choice logic approaches imply a new programming technique compared to PROLOG, because the ability to backtrack is missing. A clause in committed choice logic has the structure:

```
A :- G₁, ..., Gₘ | B₁, ..., Bₙ
```

G_1, ..., G_m are guards, and the rest is like ordinary PROLOG clauses. For reasons of efficiency, the guards are flat in the sense that they may only consist of a predefined set of (relatively simple) operations. Such a clause includes a *commit* operator `|` (omitted if the guard is empty) used to separate the right-hand side of the clause into a conjunction of guard conditions and a conjunction of subgoal predicates. The guard conditions must evaluate to `true` to enable the evaluation of the subgoals. The commit expresses *don't-care* nondeterminism, i.e., the termination of the OR-parallel evaluation of alternative clauses: if more than one clause can apply to reduce a subgoal, one is chosen arbitrarily and no backtracking can take place if that choice later results in a failure. For instance, given the following program fragment and the goal `A(0)`

```
A(X) :- X ≥ 0 | B
A(X) :- X ≤ 0 | C
```

both guards are satisfied, so one is chosen arbitrarily and `A(0)` is reduced to `B` or `C`. This fragment shows also the difference between *don't-know* nondeterminism and *don't-care* nondeterminism. If the above committed choice logic clauses were PROLOG clauses, then when the chosen clause fails by backtracking (i.e., a solution is not obtained) the other clause is later chosen. So in PROLOG, since we *do not know* which clause will be successful, all must be tried. Conversely, in committed choice logic programming we *do not care* which clause is chosen, and no backtracking is activated if a committed clause fails in the body.

The essence of don't-know nondeterminism is that failing computations *do not count* and only successful computations may produce a result. The essence of don't-care nondeterminism is that once a transition has been taken the computation is committed to it and cannot backtrack or explore in parallel other alternatives. The guards imply an important difference from PROLOG when considering OR-parallelism, since when trying a goal in parallel, only one clause is chosen and execution is then *committed* to this clause, i.e., the choice cannot be undone.

In committed choice logic, only one mechanism of communication among processes is allowed: unification of *shared logical variables*. The value of a shared logical variable can be set once only (single-assignment property). Communication in committed choice logic is associative and asynchronous. The goal

```
?- A(X), B(X).
```

defines two processes that communicate by asynchronously accessing the shared logical variable `X`. This can contain a simple value, or a complex data structure. Processes communicate by instantiation of a shared logical variable and synchronize by waiting until a shared logical variable is instantiated.

Operationally, the committed choice logic model of computation consists of a dynamic set of parallel processes, communicating by instantiating shared logical variables, synchronizing by waiting for variables to be instantiated, and making nondeterministic choices, possibly based on the availability of values of variables. This general runtime structure is the same for Flat Concurrent Prolog (FCP), Parlog, and Guarded Horn Clauses (GHC). The main differences lie in the way unification is used, and in the predefined predicates in the guards offered by each language. See [Shapiro, 1989] for a discussion of the differences.

Delta-Prolog [Cunha *et al.*, 1989] is another parallel logic language, which being based on CSP [Hoare, 1985] supports a parallel model very different from that of the above discussed parallel logic languages. Delta-Prolog offers both forms of nondeterminism: committed choice (don't care) and backtrackable choice (don't know). Synchronous event goals are the Delta-Prolog counterpart of CSP's communication operations. A Delta-Prolog program without events is a PROLOG program. A Delta-Prolog program with events maintains a PROLOG flavor, since events are backtrackable. This implies the need for a mechanism for *distributed backtracking*. When a process tries to backtrack an event goal, it must inform its partner process which also has to engage in backtracking to that event. This is necessary to guarantee a complete search in the solution space of the problem.

Bibliographic Notes

For a full account to PROLOG we refer the reader to [Clocksin and Mellish, 1987]. The first PROLOG-example of this section is from [Anderson *et al.*, 1990]. [Ciancarini, 1992] is a survey of several parallel logic languages. For a full account to parallel committed choice logic we refer to [Shapiro, 1989]. Strand [Foster and Taylor, 1989] is a commercial parallel programming system based on committed choice logic which comprises a language, a development environment and parallel programming libraries.

3.2.5 Unbounded Nondeterministic Iterative Transformations

The computational model of Unity is unbounded nondeterministic iterative transformations of the program state [Chandy and Misra, 1988]. A Unity program is essentially a declaration of a set of variables, a specification of their initial values, and a set of multiple-assignment statements. Program execution starts from any state satisfying the initial condition and consists in selecting nondeterministically some assignment statement, executing it, and repeating forever. The set of statements and variables in a Unity program is fixed at compile time. Unity is based on the array data structure. Nondeterministic selection of statements is constrained by the following fairness rule: every statement is selected infinitely often. This is unconditional fairness according to [Kwiatkowska, 1989] (see also our discussion of fairness in Sect. 5.4.4).

Unity programs terminate by reaching a *fixed point*. A program reaches a fixed point in its execution if the execution of any statement does not change the state of the program. Therefore, one way to implement termination for a Unity program is to halt it after it reaches a fixed point.

To illustrate the language, we present a solution to the problem of finding the earliest meeting time acceptable to every member of a group of people. Time is integer valued and nonnegative. We take a group of three people: F, G, and H. Associated with the persons F, G, and H are functions f, g, and h (respectively), which map times to times. The meaning of f is as follows (and the meanings of g and h follow by analogy). For any t, $f(t) \geq t$ holds; person F can meet at time $f(t)$ and cannot meet at any time u where $t \leq u < f(t)$. Note that $f(f(t)) = f(t)$ holds, and $t = f(t)$ means that F can meet at time t. Thus $f(t)$ is the earliest time at or after t at which person F can meet. We assume for simplicity that there exists some common meeting time for the involved persons. In the following program, the symbol $[\!]$ is used to separate the assignment statements. The initial condition is specified under **initially**:

```
Program Meeting
    initially   r = 0
    assign      r := f(r) [] 
                r := g(r) []
                r := h(r)
end {Meeting}
```

This program has three assignments: $r := f(r)$, $r := g(r)$, and $r := h(r)$. Computation proceeds by executing any one of the assignments, selected nondeterministically. The selection obeys the fairness rule: every assignment is executed infinitely often. Initially the proposed meeting time is zero. Any of the participants — F, G, or H — increases the value of the proposed meeting time to the next possible time, if he cannot meet at that time. At fixed point, r is a common meeting time. Fair selection is necessary to guarantee that r is the common meeting time at the fixed point.

Such a Unity program can be viewed as a restriction of Dijksta's guarded command notation [Dijkstra, 1975], allowing programs to contain only a single repetitive construct. All the guards are *true* and the commands are the assignments. However, guarded commands do not guarantee fairness.

Unity's goal is to unify seemingly disparate areas of programming with a simple theory consisting of a model of computation and an associated proof system. Its computational model is built upon a traditional imperative foundation, a state-transition system with named variables to express the state and multiple-assignment statements to express the state transitions. On top of that foundation, however, Unity follows a more radical design: all flow-of-control and communication constructs have been eliminated from the notation.

The resulting computational model is that a program generates a set of execution sequences (sequences of states). When the program is executed on a parallel machine, some statements could be executed in parallel. The Unity notation intends to avoid specifying sequential dependencies which are not inherent in the problem to be solved. Unlike sequential languages the order of execution of the assignment statements has no relation to the order in which they are written. Assignments to different variables could be executed in parallel, but assignments to the same variable must occur atomically and thus in sequence.

A Unity program describes *what* should be done in the sense that it specifies the initial state and the state transformations (i.e., the assignments). A Unity program does not specify precisely *when* an assignment should be executed — the only restriction is a rather weak fairness constraint: every assignment is executed infinitely often. Neither does a Unity program specify *where* (i.e., on which processor) an assignment is to be executed, nor to which process an assignment belongs. Also, a Unity program does not specify *how* assignments are to be executed or *how* an implementation may halt a program execution. Termination is regarded as a feature of an implementation. Concerns between *what* on one hand, and *when*, *where*, and *how* on the other are separated. The *what* is specified in a program, whereas the *when*, *where*, and *how* are specified in a mapping to a particular target architecture. We refer to [Chandy and Misra, 1988] for a description of mappings to particular target architectures in Unity.

The main objective of Unity is the systematic development of programs which may be implemented on different architectures. Program development is carried out in two basic steps: first a correct program is derived from a specification, then this program is adapted to the target architecture. This adaptation is achieved by successive transformations of the original program in order to make control explicit. The multiple-assignment statement is used to express the mapping onto synchronous shared-memory architectures, and the mapping onto asynchronous architectures is achieved by partitioning the statements of the program. Unity uses sets of assignment statements to avoid overspecification of control flow, and maps from statements to processors in order to specialize an algorithm to a particular topology.

Bibliographic Notes

[Chandy and Misra, 1988] describe the language Unity and its associated proof system. The problem and solution of finding the earliest meeting time acceptable to every member of a group of people is from this book.

We find a fundamental difference between Unity and committed choice logic languages (Sect. 3.2.4) in the notion of a variable. In Unity, variables are mutable; therefore, transitions must exclude other transitions from writing variables they read from, and from accessing variables they write to. In committed choice logic languages, variables are single-assignment; therefore, no mutual exclusion mechanisms are required when reading a variable. However, the Parallel Program Design approach of [Chandy and Taylor, 1992] is derived in part from the Unity model and the model of committed choice logic.

3.2.6 Multiset Transformations

The GAMMA formalism [Banâtre and Métayer, 1993] has been proposed to allow the description of programs without artificial sequentiality. Sequentiality is considered artificial when it is not implied by the logic of the program. GAMMA is a minimal language based on one data structure, the multiset, and the corresponding control structure, the *chemical reaction*.

Therefore, GAMMA implements the idea of a chemical reaction of the elements in a multiset. Some elements which fulfill a certain predicate (also called *reaction condition*) may be taken from the multiset and replaced by new elements which are generated by combining the selected elements using the so-called *reaction function*. For example, a very simple GAMMA program computing the largest set S included in a multiset M is (written in guarded commands like syntax):

```
$S := M;$
do
   $\exists\, x \in S, \exists\, y \in S - \{x\}, x = y \longrightarrow S := S - \{x\}$
od
```

An intuitive way of describing the meaning of a GAMMA program is the metaphor of the chemical reaction: the multiset can be seen as a chemical solution. The guard ("$\exists\, x \in S, \exists\, y \in S - \{x\}, x = y$" in our example) is the reaction condition which is a property to be satisfied by reacting elements. The action ("$S := S - \{x\}$" in our example) describes the product of the reaction. The computation terminates when a stable state is reached, that is to say when no elements of the multiset satisfy the reaction condition. In GAMMA, multisets are enclosed within $\{\}$ since GAMMA does not support sets.

The above example also serves as an illustration of the differences between sets and multisets. As another example, consider the following program which returns the set of prime numbers smaller than number N:

```
prime_numbers(N) = P({2, ..., N})  where
                                   P(M) =  do
                                              replace x, y ∈ M
                                              if multiple (x,y)
                                              by y
                                           od
```

This program proceeds by removing from the multiset $\{2, \ldots, N\}$ elements that are multiples of an other element in the multiset. The function *multiple (x,y)* yields **true**, if x is a multiple of y.

If several reaction conditions hold for several subsets of a multiset at the same time, the choice which is made among them is nondeterministic. As a consequence, if the reaction condition holds for several disjoint subsets, the reactions can be carried out independently (and simultaneously). This property is the basic reason why GAMMA programs do generally exhibit a lot of potential parallelism. Since all reactions operate on disjoint elements, they can take place simultaneously, and GAMMA programs can be naturally executed in a data-parallel way.

GAMMA is not intended as a programming language in the usual sense of the term. It is designed as an intermediate language between specifications and programs: it is possible to express in GAMMA the idea of an algorithm without any detail about the execution order or the memory management. GAMMA programs are executable but any straightforward implementation would be extremely inefficient. These features make GAMMA a candidate for prototyping parallel algorithms.

Bibliographic Notes

The program computing the largest set included in a multiset is from [Mussat, 1992] and the program which returns the set of prime numbers is from [Banâtre and Métayer, 1990].

The chemical abstract machine model [Berry and Boudol, 1990] is based on the concepts of GAMMA. It elaborates the original GAMMA language by presenting *molecules* in a systematic way as terms of algebras, and refining the classification of rules. Molecules encapsulate subsolutions.

3.2.7 Virtual Shared Memory

Virtual shared memory in its most general sense refers to a provision of a shared address space on distributed memory hardware. It intends to combine the scalability of network-based architectures with the convenience of shared-memory programming. The shared-memory abstraction gives these systems the illusion of physically shared memory and allows programmers to use the shared-memory paradigm. Such architectures contain no physically shared memory. Instead the distributed local memories collectively provide a virtual address space shared by all the processors. Virtual shared memory is, therefore, essentially an *emulation* of shared memory on a distributed memory parallel computer. It combines the benefits of the ease of programming found in shared-memory multiprocessors with the scalability of message-passing multiprocessors. The implementations of virtual shared memory vary from software-based schemes, integrated into the operating system, to hardware-based schemes which employ conventional caching algorithms. Virtual shared memory systems have been implemented using three approaches (some systems use more than one approach):

1. hardware implementations that extend traditional caching techniques to scalable architectures,
2. operating system and library implementations that achieve sharing and coherence through memory-management mechanisms, and
3. compiler implementations where shared accesses are automatically converted into synchronization and coherence primitives.

The address space in virtual shared memory systems is usually divided into fixed-size pages, which are distributed among the processors. Processes either have *no*, *read* or *read/write* access to a page. *Read* pages can be replicated on multiple processors to reduce access times. Mutual-exclusion synchronization can be implemented by locking pages.

A conventional virtual memory system *pages* data between the main memory of a processor and disks. A virtual shared memory system *pages* data between the main memories of individual processors. The size of the pages is an important consideration to gain acceptable performance. A suitable compromise in granularity is the typical *page* used in conventional virtual memory implementations [Li and Hudak, 1989].

The shared data-object model and the model of generative communication can be regarded as *structured* approaches to virtual shared memory. We shall discuss these models in Sects. 3.2.8 and 3.2.9, respectively.

Bibliographic Notes

For comparative studies of virtual shared memory systems we refer to [Nitzberg and Lo, 1991; Raina, 1992]. Besides being referred to as virtual shared memory, such architectures have also been given different names such as *shared virtual memory*, *distributed shared memory*, or *distributed shared virtual memory*. [Hill *et al.*, 1992] call it *cooperative shared memory* which is implemented as a combination of software and hardware intended to provide scalable shared memory.

A critical question for virtual shared memory is: How to retain the consistency on a physically distributed environment? We refer to [Stumm and Zhou, 1990] for a good evolutionary overview of algorithms implementing virtual shared memory. These algorithms support static, migratory, and replicated data.

3.2.8 The Shared Data-Object Model

The shared data-object model is a model for distributed programming using shared data. The procedural, statically typed programming language Orca is based on the shared data-object model [Bal *et al.*, 1992]. It is designed for distributed, non-shared memory systems, but provides logically shared data to the programmer (similar to virtual shared memory, but in a structured way). Pointers have intentionally been omitted in Orca to provide security. Also the language lacks global variables. Instead, variables can be passed as reference parameters.

Orca only allows processes to share data-objects of *abstract data types*. Data-objects are best thought of as instances (variables) of abstract data types. A data-object is a passive entity: it only contains data and no process. Such an object can only be manipulated by the operations defined by its abstract data type. Consequently, access to shared data in Orca always occurs through user-defined, high-level operations. The shared data-object model is characterized by two important rules:

1. each operation is executed indivisibly, and
2. each operation is applied only to a single object.

Executing operations indivisibly simplifies programming, since the programmers do not have to worry about mutual exclusion synchronization. Conceptually all operations on a given object are serialized. However, an optimizing programming system may execute operations in parallel, provided that it obtains the same effect as serialized operations.

Operations on multiple shared data-objects are not supported to allow an efficient implementation. Such operations can be implemented on top of the shared data-object model, although the programmer has to take care of synchronization in this case.

Mutual exclusion synchronization in Orca is done implicitly by executing all operations on objects indivisibly. Condition synchronization is integrated in the model by allowing operations to block (wait) until a certain condition becomes true. An operation is only allowed to block initially. Such an operation consists of a guarded statement, containing conditions (Boolean expressions) and statement lists, very similar to guarded commands. The operation blocks until at least one of the conditions is true; next, it chooses one of the guarded statements whose condition is true and executes the corresponding statement list, without blocking again. Blocking operations in nested objects require a somewhat complicated execution model (see [Bal *et al.*, 1992] for a detailed discussion).

An important semantical issue concerns the ordering of operations on shared data-objects. The model supports a *consistent ordering* of operations: it guarantees that all processes will observe operations on shared objects in the same order.

An abstract data type definition in Orca consists of two parts: a specification part and an implementation part. The specification part gives the operations that can be applied to variables (objects) of this type. The implementation part contains the data used to represent variables of this type, the implementation code of the operations, and code for initializing variables of the type.

To illustrate the language, Fig. 3.1 gives an example of the specification part of a simple abstract data type, encapsulating a single integer variable. Fig. 3.2 shows the implementation part which contains the data used to represent objects of this type, the code to initialize the data of new instances of the type, and the code implementing the operations. An operation implementation is similar to a procedure. An operation can only access its own local variables and parameters, and the local (internal) data of the object it is applied to.

Parallelism is expressed in Orca through sequential processes. Processes can communicate through shared data-objects. An object may be passed as a shared parameter to a child process, as indicated by the following declaration of the `child` process:

```
process child (X: shared IntObject);
begin ...  end;
```

```
object specification IntObject;
   operation Value(): integer;       # return current value
   operation Assign (val: integer);  # assign a new value
   operation Add (val: integer);     # add val to the current value
end;
```

Figure 3.1: Specification part of an abstract data type in Orca.

```
object implementation IntObject;
   x: integer;                       # internal data

   operation Value(): integer;
   begin
         return x;                   # return current value
   end;

   operation Assign (v: integer);
   begin
         x := v;                     # assign a new value
   end;

   operation Add (v: integer);
   begin
         x := x+v;                   # add v to the current value
   end;
begin
      x := 0;                        # initialize to zero
end;
```

Figure 3.2: Implementation part of an abstract data type in Orca.

A process can be created dynamically by a **fork** statement. We can declare an object **MyObj** of abstract data type **IntObject** and pass this object as a *shared* parameter when creating a new child process:

```
MyObj:  IntObject;
...
fork child (MyObj);
```

This is similar to calling a conventional procedure and passing a *call-by-reference* parameter to it, except that the parent and child will execute in parallel. Any number of child processes can be created in this way, and the children may pass the shared objects on to their children, and so on. So a hierarchy of processes communicating through shared data-objects can be created. The language does not provide sharing of data between independent processes (i.e., without a common ancestor).

The model is implemented by replicating objects. The idea is to store a local copy of an object wherever it is needed, thus read operations on the object can be done locally. An important issue is how to update the replicas of an object to guarantee the consistent ordering of the operations. The implementation must guarantee this ordering, despite using replication. [Bal *et al.*, 1992] use *reliable indivisible broadcasting* to guarantee consistent ordering in one implementation and a two-phase update protocol in another implementation which uses remote procedure calls.

Bibliographic Notes

The shared data-object model has been developed by Henri Bal at the Vrije Universiteit in Amsterdam [Bal, 1990]. The example in Fig. 3.1 and Fig. 3.2 is from [Bal *et al.*, 1992]. A new implementation which uses either full or no replication of objects is presented in [Bal and Kaashoek, 1993]. The decision to replicate objects is based on an integration of compile-time and run-time analysis.

3.2.9 Generative Communication

This section presents the model of generative communication with the coordination language Linda [Gelernter, 1985]. Since we took Linda's concepts as one essential basis for our work, we shall discuss it with greater detail than the previously presented languages.

Linda is a coordination language concept for explicitly parallel programming in an architecture independent way. Communication in Linda is based on the concept of tuple space, i.e., a virtual common data space accessed by an associative addressing scheme. Process communication and synchronization in Linda is reduced to concurrent access to a large data pool, thus relieving the programmer from the burden of having to consider all process inter-relations explicitly. The parallel processes are decoupled in time and space in a very simple way: processes do not have to execute at the same time and do not need to know each other's addresses. This scheme offers all advantages of a shared memory architecture, such as anonymous communication and easy load balancing. It adds a very flexible associative addressing mechanism, a natural synchronization paradigm and at the same time it avoids the well-known access bottleneck for shared memory systems as far as possible.

The shared data pool in the Linda concept is called *tuple space*. Its access unit is the tuple, similar to tuples in ProSet (Sect. 4.1). Tuples live in tuple space which is simply a collection of tuples. It may contain any number of copies of the same tuple: it is a multiset, not a set. Tuple space is the fundamental medium of communication. All Linda communication is a three-party operation: sender interacts with tuple space, tuple space interacts with receiver. Conversely, traditional models such as message passing provide two-party operations.

Process communication and synchronization in Linda is called *generative communication*, because tuples are added to, removed from, and read from tuple space concurrently. Synchronization is done implicitly.

Reading access to tuples in tuple space is associative and not based on physical addresses — in fact, the internal structure of tuple space is hidden from the user. Reading access to tuples is based on their expected content described in so-called *templates*. This method is similar to the selection of entries from a data base. Each component of a tuple or template is either an *actual*, i.e., holding a value of a given type, or a *formal*, i.e., a declared placeholder for such a value. A formal is prefixed with a question mark. Tuples in tuple space are selected by *matching*, where a tuple and a template are defined to match, iff they have the same structure (corresponding number and type of components) and the values of their actuals are equal to the values of the corresponding tuple fields.

The combination of the computation language C with the coordination language Linda forms the parallel programming language C-Linda [Carriero and Gelernter, 1992]. C-Linda defines six operators, which may be added to a sequential computation language. These operators enable sequential processes, specified in the underlying computation language, to access the tuple space. Of the six operations, two produce tuples and four consume them:

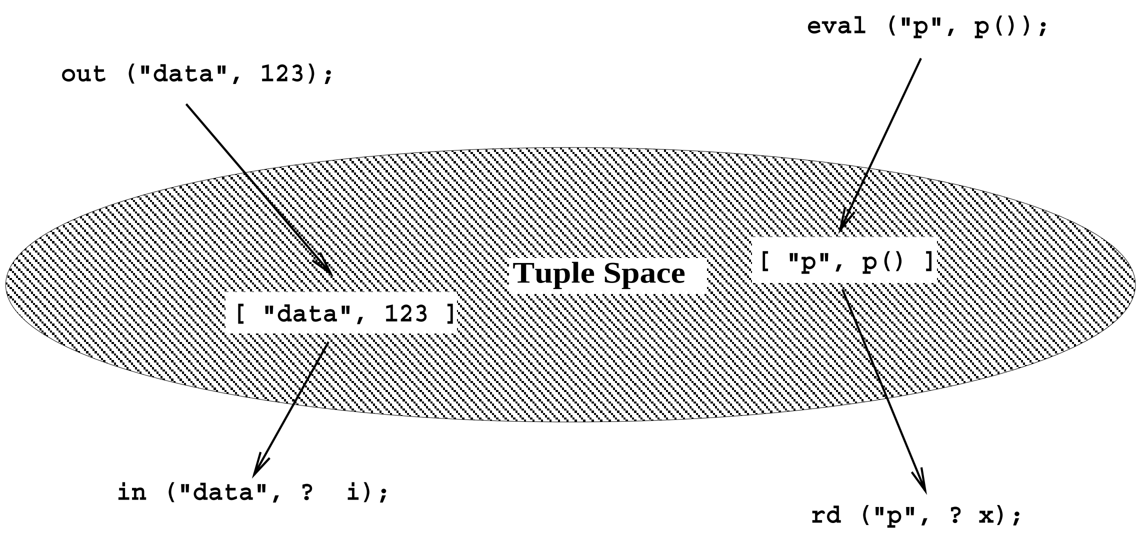


Figure 3.3: Tuple-space communication in C-Linda.

Generative Operations

`out(tuple);` The specified tuple is evaluated and then added to the tuple space. The `out`-executing process continues as soon as the evaluation of the tuple is completed. The "`out("data", 123);`" operation in Fig. 3.3 deposits the tuple `["data",123]` into tuple space.

No specified action is taken in the event that tuple space is full. We will discuss in Sect. 7.6 the notion of *full* tuple spaces.

`eval(tuple);` Executing an `eval` operation causes the following sequence of activities. First, bindings for names indicated explicitly in the tuple are established in the environment of the `eval`-executing process. At this point, the `eval`-executing process may continue. Each field of the tuple argument to `eval` is now evaluated, independently of and asynchronously with the `eval`-executing process and each other. The fields of an `eval` tuple are evaluated concurrently yielding one thread of execution for every field. `eval` deposits *active* tuples into tuple space, which are *not* accessible to the remaining four operations. Conversely, `out` deposits *passive* tuples into tuple space, which are accessible to the remaining four operations, which are discussed below. When every field has been evaluated completely, the tuple consisting of the values yielded by each `eval`-tuple field, in the order of their appearance in the `eval` tuple, becomes available in tuple space: the active tuple converts to a passive one. The "`eval("p",p());`" operation in Fig. 3.3 deposits the active tuple `["p",p()]`, containing two processes, into tuple space.

Some current implementations in fact evaluate all fields of an `eval` tuple sequentially within a single new process. This may cause deadlocks if processes within an `eval` tuple communicate with each other. The Yale Linda Implementation would only spawn one process in the above example, since only expressions consisting of a single function call are evaluated within *new* processes by this implementation [Carriero and Gelernter, 1990a].

The main program is the only process that lives outside the tuple space in C-Linda.

Blocking Extraction Operations

`in(template);` The `in` operation attempts to withdraw a specified tuple from tuple space. Tuple space is searched for a matching tuple against the template supplied as the operation's argument. When and if a tuple is found, it is withdrawn from tuple space, and the values of its actual fields are bound to any corresponding formals in the template. Tuples are withdrawn *atomically*: a tuple can be grabbed by only one process, and once grabbed it is withdrawn entirely. If no matching tuple exists in tuple space, the process executing the `in` suspends until a matching tuple becomes available. If many tuples satisfy the match criteria, one is chosen arbitrarily. The "`in("data",?i);`" operation in Fig. 3.3 withdraws the tuple `["data",123]` from tuple space and assigns `123` to the integer variable `i`.

`rd(template);` The `rd` operation is the same as `in`, with actuals assigned to formals as before, *except* that the matched tuple remains in tuple space. The "`rd("p",?x);`" operation in Fig. 3.3 has to wait for the termination of `p()` to read the return value of `p()`. It is presupposed that the return value of `p()` has the same type that the variable `x` is declared with.

Non-blocking Extraction Operations

`inp(template) / rdp(template)` These operations attempt to locate a matching tuple and return `0` if they fail; otherwise, they return `1` and perform actual-to-formal assignment as described above. The only difference with `in/rd` is that the predicates will not block if no matching tuple is found.

It may be difficult to implement these operations on distributed memory architectures (see also Sect. 7.6 for a discussion on this subject).

To summarize, a tuple and a template match in C-Linda iff

- the tuple is passive,
- the numbers of fields are equal,
- types and values of actuals in templates are equal to the corresponding tuple fields, and
- the types of the variables in the formals are equal to the types of the corresponding tuple fields.

C-Linda allows formals in deposited tuples. Such formals match with appropriate actuals in templates, but never with formals in templates (see also Sect. 7.9 for a discussion on this subject).

Bibliographic Notes

Linda has been developed by David Gelernter at Yale University [Gelernter, 1985]. We refer to [Carriero and Gelernter, 1990a] for a full account to parallel programming in Linda. Our description of the C-Linda operations is derived from [Carriero and Gelernter, 1990a, Appendix A]. Comparisons of Linda with other approaches to parallel programming may be found in [Carriero and Gelernter, 1989; Bjornson *et al.*, 1991; Matrone *et al.*, 1993].

A *coordination language* like Linda provides means for process creation and inter-process communication which may be combined with *computation languages* like C [Carriero and Gelernter, 1992]. A *parallel programming language* consists, therefore, of a coordination language and a sequential computation language. In Linda, coordination and computation are two separate issues of equal standing which together define the problem of building software. The first computation language, in which Linda has been integrated, is C [Gelernter *et al.*, 1985; Ahuja *et al.*, 1986; Carriero *et al.*, 1986; Bjornson *et al.*, 1988]. Meanwhile there exist also integrations into higher-level languages such as

C++ [Callsen *et al.*, 1991]

Eiffel [Jellinghaus, 1990]

EuLisp [Broadbery and Playford, 1991]

Joyce [Pinakis and McDonald, 1991]

Lisp [Hutchinson, 1990; Abarbanel, 1991; Yuen *et al.*, 1993]

Maple [Char, 1990]

ML [Siegel and Cooper, 1991]

Modula-2 [Borrman *et al.*, 1989; Trescher *et al.*, 1992; Pouget and Burkhart, 1993]

PostScript [Leler, 1989]

PROLOG [Sutcliffe and Pinakis, 1990; Anderson *et al.*, 1990; Ciancarini, 1991; Bosschere and Wulteputte, 1991; MacDonald, 1991; Sutcliffe, 1993; Bosschere *et al.*, 1993]

Russell [Butcher and Zedan, 1991]

Scheme [Dourish, 1989; Dahlen and MacDonald, 1990; Jagannathan, 1991; Wack, 1993]

Smalltalk [Christiansen *et al.*, 1987; Matsuoka and Kawai, 1988; Peskin and Segall, 1991]

Combinations with operating systems are reported in [Fleckenstein and Hemmendinger, 1989; Leler, 1990]. The combination with Joyce [Pinakis and McDonald, 1991] is remarkable, because Joyce is already a parallel language with support for CSP-like synchronous channels for interprocess communication. In Joyce-Linda the synchronous channels were discarded in favor of the more general Linda operations.

Linda supports the master-worker model with distributed data structures: one master process interacts with a collection of identical workers. The master generates task tuples and collects results while the worker processes repeatedly grab tasks from tuple space, perform the required actions, and return result tuples to tuple space. This model allows easy load balancing. See also Chap. 6 for examples. A special instance of the master-worker model is the so-called *Piranha model* for LAN-connected workstations, where computational piranhas attack a *cloud of tasks* [Carriero *et al.*, 1993].

Programming environments for Linda have been built. TupleScope [Bercovitz and Carriero, 1990] is a graphical visualization and debugging tool for C-Linda programs. Another graphical debugger/monitor for Linda is presented in [Sewry, 1991]. [Ahmed and Gelernter, 1992] present the Linda Program Builder, an Epoch-based editor (Epoch is a multi-windows version of Emacs running under X-Windows). This system supports incremental development of C-Linda programs. The Linda Program Builder supports programming paradigms that underlie most parallel programs such as the master-worker model and the use of distributed data structures.

Also hardware support (see Sect. 11.3.3) and fault tolerance (see Sect. 7.14) have been considered for implementations of Linda. Some applications experience is discussed in [Carriero and Gelernter, 1988; Bjornson *et al.*, 1991; Smith, 1991; Cannon, 1992].

There exist some approaches which have features in common with Linda's concept for coordination via tuple space. Ease [Zenith, 1992] combines the ideas of Linda and of message passing. So-called *contexts* are the counterparts of channels in message passing, and Linda-like operations on such contexts are allowed. However, the operations on these contexts are restricted to increase the efficiency in a way that no runtime matching is required. These restrictions reduce the flexibility for building distributed data structures significantly. The ActorSpace model [Agha and Callsen, 1993] combines the actor model (Sect. 3.2.2) with pattern-based communication which is very similar to tuple-space communication.

The Swarm model [Roman and Cunningham, 1990] merges the philosophy of Unity (Sect. 3.2.5) with the methods of Linda. Swarm has a Unity-like program structure and the computational model of Linda-like communication mechanisms. The model partitions its *dataspace* into three subsets: a tuple space, a transaction space (a finite set of transactions), and a synchrony relation (a symmetric relation on the set of all possible transactions). Swarm replaces Unity's fixed set of variables with the tuple space and Unity's fixed set of assignment statements with a set of transactions. The analogue of executing a statement is executing a transaction which involves making a nondeterministic choice between all tuples that satisfy a given criterion. The transaction construct reduces both communication and synchronization to the notion of an atomic transformation of the dataspace. However, in Swarm there is no concept of a process, and there are no sequential programming constructs. Sequencing is accomplished by defining *continuations* in the form of new transactions to be inserted into the dataspace.

3.2.10 Evaluation

We studied various models for parallel programming in the preceding subsections. Our study started with data-parallel approaches followed by extensions to object-oriented, functional, and logical languages. We continued with the fundamentally new concepts of unbounded nondeterministic iterative transformations and multiset transformations, and proceeded via virtual shared memory to the shared data-object model and generative communication. We will now evaluate these models concerning their suitability for prototyping parallel algorithms in a set-oriented language.

Message Passing

The message-passing model envisions a collection of parallel activities, each resembling a self-contained, sequential process. Each of these processes consists of program structures and data structures. The program structures are active: they execute instructions on the data structures.

Processes that are collaborating on a problem will ordinarily need to share data, but in the message-passing model data structures are sealed within processes, and so processes cannot access the others data directly. Instead they exchange messages. When one process has data for another one, it generates a message and hands it to a process of a different kind, a message-delivery process. The message-delivery process routes the message to its destination. This scheme adds complexity to the program as a whole: it means that each process must know how to generate messages and where to send them. We refer to [Bal, 1990] for an extensive discussion of the shortcomings of the message-passing model.

In contrast to the message-passing model, the shared-memory model allows application programs to use shared memory as they use normal local memory. The primary advantage of shared memory over message passing is the simpler abstraction provided to the application programmer, an abstraction the programmer already understands well. This allows a more natural transition from sequential to parallel programming. Message passing is less suitable when several processes need to coordinate indirectly by sharing global state information.

The message-passing model forces the programmer to be conscious of data movement between processes at all times, since processes must explicitly use communication primitives and channels or ports. Consequently, the code written with the shared memory model is usually significantly shorter and easier to understand than equivalent programs that use message passing.

Most of the problems which arise with message passing exist similarly for graphical approaches based on Petri-nets or data-flow diagrams, because such graphs can be regarded as graphical representations of message-passing systems.

Data Parallelism

Data parallelism is opposed to *control parallelism* which is achieved through multiple threads of control, operating independently. The data parallel approach lets programmers replace iteration (repeated execution of the same set of instructions with different data) with parallel execution. It does not address a more general case, however: performing many interrelated but *different* operations at the same time. This ability is essential in developing complex application programs.

Let us take a closer look at Parallel SETL [Hummel *et al.*, 1991]. Compare the parallel iteration:

```
for i par_over {1..1000} do
     statements
end do;
```

with a similar sequential iteration:

```
for i in {1..1000} do
     statements
end do;
```

Both loops have the same expressiveness concerning the expression of algorithms, since in both cases the iteration order is not over-specified. A compiler could replace the `in` by `par_over` as FORTRAN-compilers automatically try to parallelize DO-loops (provided that no global variables are changed in the enclosed statements). Therefore, we regard the `par_over` as an aid to a compiler. This construct indicates where it pays to parallelize an iteration. Thus the motivation for replacing the `in` by `par_over` is to increase the execution performance and not the expressiveness of the program. The `in` is already implicitly a parallel iteration.

We are searching for appropriate means that enable the expression of *new* parallel algorithms for implementing inherently parallel systems, and not primarily to increase the execution performance of prototypes.

Our previous work was concerned with developing a language for data-parallel image processing [Hasselbring, 1990]. As the underlying model of this approach uses synchronous communication, the programmer often has to think in *simultaneities* while constructing a program, because she or he often has to focus on more than one process at a time.[1] As noted earlier, this complicates parallel programming significantly. A more detailed comparison of this data-parallel approach with generative communication may be found in [Hasselbring, 1992b].

Parallel Object-Oriented Programming

Parallel object-oriented languages tend to use either message passing or remote procedure calls for inter-process communication: the object space (the collection of all objects in the program) is not the communication medium, and does not constitute a shared object memory in our sense. Additionally, as mentioned in Sect. 3.2.2, probably the most difficult aspect of integrating parallelism into object-oriented languages is that inheritance greatly complicates synchronization.

Although, in some sense, parallel object-oriented languages allow processes (usually objects) to share data (also objects), their spirit is closer to message passing than to shared variables. Therefore, most of the problems which arise with message passing exist similarly for parallel object-oriented languages.

Parallel Functional Programming

Pure functional languages are not suitable for programming cooperating processes: they are deterministic and they do not have variables. Therefore, processes described as functions cannot include choices of alternative actions and they cannot remember their states from one action to another. Nondeterminism would destroy referential transparency in functional programming languages.

Processes sometimes cooperate in a way that cannot be predicted. It is impossible, for instance, to predict from which terminal of a multi-user computing system the next request for a particular service might come. Moreover, the system behavior necessarily depends on previous requests.

Both nondeterminism of events and dependence on the process history are strong arguments for an imperative rather than applicative programming model for cooperating processes. This is due

[1]In the remainder of this thesis we will not distinguish between males and females. When we say that *he* could do something, we mean that *she or he* could do something.

to the determinism and the lack of variables which make pure functional languages impractical for programming parallel systems. Even the parallel functional languages that are presented in Sect. 3.2.3 do not support nondeterminism.

Futures in Multilisp provide a method for process creation but no means for synchronization and communication between processes, except for waiting for each other's termination. However, process creation via futures provides a high-level abstraction for achieving parallel execution.

Parallel Logic Programming

In [Shapiro, 1989], it is described how several communication patterns can be expressed using shared logical variables despite the single-assignment property of such variables. On the other hand, the shared logical variable also has its problems. Although it is possible to implement shared data structures like streams and queues using shared logical variables, only a single process can add elements to such data structures. See [Gelernter, 1984] for a discussion of the problems thus caused.

The underlying idea in the family of committed choice parallel logic languages is to model synchronization between processes by imposing some constraints on the unification mechanism. Notably, the committed choice logic languages are usually *flat* languages, i.e., the predicates in the guards cannot be used to make computations. We quote for the problems thus caused:

> "Since the guards cannot be used to make computations, a deterministic style of programming is encouraged, so that OR-parallelism is usually greatly reduced. [...] From the programmer's point of view, unification is a powerful mechanism. Unfortunately, the special constraints and idiosyncrasies embedded in the different committed choice logic languages make them difficult to use in practice. In fact, all these languages result verbose and difficult to use for the specification of systems where many-to-one communication is needed (for example, many clients querying a database). [...]
>
> Another important drawback is that, although great effort has been devoted to clarify the mathematical aspects of these languages, their practical use cannot avoid relying on a number of predefined predicates whose formal semantics is often obscure." [Ciancarini, 1992, pages 234–235]

Note that the committed choice logic languages which have abandoned backtracking require a different style of programming compared to PROLOG.

The Delta-Prolog [Cunha *et al.*, 1989] parallel programming techniques are based on CSP. This implies that the problems with message passing exist for Delta-Prolog accordingly. Delta-Prolog's main special feature lies in the fact that communication is backtrackable. This is a very heavy requirement for its formal semantics and for an implementation; moreover this feature does not seem to be really useful for increasing language expressiveness. For instance, backtrackable input/output is rarely needed in practice [Ciancarini, 1992].

Unbounded Nondeterministic Iterative Transformations

Unity uses sets of assignment statements to avoid forcing programmers to overspecify control flow, and maps from statements to processors in order to specialize an algorithm to a particular topology. However, a Unity program has no control flow, consisting of a collection of instructions that are executed infinitely often. We believe that abandoning control flow altogether is too drastic: programmers are accustomed to imperative languages, and moreover, programs generally consist of both sequential and parallel parts. We prefer a model that allows for a gradual transition from serial to parallel algorithms.

Additionally, Unity is static. No dynamic process creation and termination is possible. Computations are defined by a fixed set of statements and variables. Unity intends to model the execution of programs on various kinds of architectures. We intend to model the execution of programs on a high level without considering specific architectures. Unity pays the price of less elegant solutions for the advantage of a unified framework which leads to a restricted notation.

Multiset Transformations

The Gamma approach is very similar to that of Unity. The main difference between Gamma and Unity is that the latter is based on the array data structure, which makes the treatment of dynamically varying size problems less natural, whereas the former is based on the more flexible multiset data structure.

As noted in Sect. 3.2.6, the concept of multiset transformations in Gamma is a candidate for prototyping parallel algorithms. However, some solutions seem inherently difficult to express in Gamma. We quote from the designers of Gamma:

> "When constructing a Gamma program, the challenge consists in expressing computation as a collection of local operations. This unusual view of programming may sometimes lead to surprising programs, but some problems turn out to be very difficult to cast into this framework. The only known solutions to certain problems rely on control decisions involving the whole state of computation. [...] Of course, these problems can be solved in Gamma, but their expression is inelegant because the control has to be encoded within the multiset." [Banâtre and Métayer, 1993, page 108]

Additionally, parallelism is implicit rather than explicit in Gamma. However, we regard multisets as a very powerful and flexible device for the coordination of parallel processes.

Virtual Shared Memory

As we mentioned earlier, it is generally accepted that programming shared-memory multiprocessors is easier than programming distributed-memory systems. The virtual shared memory approach intends to combine the advantages of both: the scalability of distributed memory, and the programmability of shared memory. However, virtual shared memory is accessed through low-level read and write instructions, and shared memory is partitioned into fixed-size pages. Therefore, virtual shared memory is a somewhat low-level realization of the shared memory abstraction. The granulatity of the shared data is fixed in virtual shared memory systems and is not user-defined.

The Shared Data-Object Model

Orca supports user-defined, high-level operations on shared data-objects. The granulatity of the shared data is chosen by the programmer. However, in the shared data-object model, processes are always organized in a hierarchy. Orca does not provide sharing of data between independent processes. Conversely, the programmer is not restricted to hierarchies with generative communication. He can organize the processes as he sees fit. The access to tuple space is associative. This allows a high degree of decoupling between the cooperating processes. Conversely, Orca's data-objects are accessed by directly passing as a parameter. This implies a tight coupling.

For cooperating processes, a flexible device for communication is needed. Access to shared data-objects in Orca is synchronized through critical regions on these objects: the operations are executed indivisibly. Therefore, the access to shared data in Orca is competing rather than cooperating. Conversely, our goal is to provide mechanisms for parallel processes to solve problems cooperatively. See also our discussion on multiprocessing and multiprogramming in Sect. 7.15.

Generative Communication

Parallel programming is conceptually harder to undertake and to understand than sequential programming, because a programmer often has to focus on more than one process at a time. Programming in Linda provides a spatially and temporally unordered bag of processes. Linda's global, associative object memory supports a highly *uncoupled* programming style in which processes remain mutually anonymous. Each task in the computation can be programmed (more-or-less) independently of any other task. This enables the programmer to focus on one process at a time thus making parallel programming conceptually the same order of problem-solving complexity as conventional, sequential programming. With generative communication each access to shared data is asynchronous: sender and receiver of a tuple do not have to exists at the same time and do not have to do things synchronously.

The uncoupled and anonymous inter-process communication in Linda is in general not directly supported by the target architectures. However, a *high-level* language must be able to reflect a particular top-down approach to building software, and not a particular machine architecture. This is also important for supporting portability across different machine architectures. Linda has been implemented on a wide variety of parallel architectures: shared-memory multi-processors as well as on distributed memory architectures (see also Sect. 11.1).

Programming with distributed data structures leads to a methodology in which low-level process synchronization concerns are abstracted to high-level algorithmic design issues involving data structure access and generation. It is easy, using Linda, to build the distributed data structures on which the application depends. Unlike messages in a message-passing program, passive tuples are integral parts of a tuple-space program: they hold its shared data structures. Because tuple space makes it easy for tasks to be divided on the fly and for a program's shape to change as it executes, it is a promising basis for representing complex and rapidly evolving systems.

Tuple space is perhaps best considered as a memory abstraction that may be used in many programming languages. Tuple space provides the abstraction of a shared, content-addressable memory which can be accessed by any process with equal ease. Both scheduling and communication are handled by the system. Programmers need only specify dependencies.

Multisets are a powerful data structure for parallel programming. A tuple space is a multiset of tuples and not a set of tuples. A multiset may contain multiple copies of a tuple, whereas in a set each element exists exactly once. Because of concurrent access by the cooperating processes to tuple space, it is necessary to have multisets and not sets for coordination: it is not easy and even not desired for coordination to guarantee the uniqueness of tuples. Multisets are, therefore, a good basis for communication between cooperating processes, because the data flow is not restricted unnecessarily. Furthermore, multisets are dynamic data structures that alleviate the treatment of dynamically varying size problems. The benefit of using multisets is the possibility of describing compound data without any form of constraint or hierarchy between its components. This is also the case for sets, but not for data structures such as lists which impose an ordering on the examination of the elements. Consequently, multisets allow a high degree of parallelism for cooperating processes. Advanced data structuring facilities, such as multisets, have the potential to simplify parallel programming.

However, generative communication — as it is realized in C-Linda and many other variants of Linda — is not without its shortcomings. We will discuss in Chap. 7 the drawbacks thus caused, such as missing support for information hiding, and how we propose to overcome them. This chapter on parallel programming discusses related work concerning parallel programming. Chapter 7 will discuss related work concerning generative communication.

At the beginning of this section, we stated two issues which must be addressed in designing a language for parallel programming. For extending a set-oriented prototyping language for parallel programming, we have chosen the following approaches for parallel execution and coordination:

- Parallel execution is achieved by adaptation of the concept for process creation via Multilisp's futures to set-oriented programming.

- Coordination between cooperating parallel processes is achieved by extending Linda's concept for synchronization and communication via tuple space.

These two items are our answers to the two questions which were raised at the beginning of Sect. 3.2. This concept for generative communication in a set-oriented prototyping language is the subject of Part II of this thesis.

Allowing users to define their own high-level operations on shared data has many advantages, both for the ease of programming and for the implementation. Orca, for example, supports user-defined, high-level operations on shared data-objects, but C-Linda uses a fixed number of built-in tuple-space operations to generate, read and delete tuples in tuple space, and does not allow users to define their own high-level operations on shared tuples. We shall overcome this shortcoming by enhancing generative communication with the possibility of changing tuples in tuple space. Furthermore, our new approach to integrating futures and generative communication into a prototyping language extends the basic Linda model with multiple tuple spaces, the notion of limited tuple spaces, selection and customization for matching, specified fairness of choice, and other useful features. Linda and ProSet both provide tuples thus it is quite natural to combine set-oriented programming with generative communication on the basis of this common feature to form a tool for prototyping parallel algorithms.

3.3 Summary

Parallel programming constitutes the second part of the setting for this thesis. We evaluated several approaches to high-level parallel programming concerning their suitability for prototyping parallel algorithms in a set-oriented language. Therefore, this chapter serves as a motivation for our approach to parallel programming in Part II of this thesis and as a discussion of related work concerning the design of parallel programming languages.

Part II

Generative Communication in Set-Oriented Prototyping

Chapter 4

The Prototyping Language ProSet

ProSet is an acronym for Prototyping with Sets. This set-oriented prototyping language is a successor to SETL [Schwartz *et al.*, 1986; Doberkat and Fox, 1989]. ProSet is actually under development at the University of Essen [Doberkat *et al.*, 1992a; Doberkat *et al.*, 1992b; Franke *et al.*, 1993]. The kernel of the language was at first presented in [Doberkat *et al.*, 1990a] and the system in [Doberkat *et al.*, 1990b]. As ProSet is a successor to SETL, it was formerly called SETL/E. The name was changed to emphasize the prototyping aspect. A first implementation is described in [Doberkat *et al.*, 1992c]. The compiler is available from the University of Essen. A typical application domain for ProSet is the evolutionary development of functional core layers of software systems. The language is not intended for prototyping graphical user interfaces.

The following sections will present a brief introduction to data and control structures of the language and two introductory examples. The advanced features for exception handling, persistence and modules are sketched. For a full account to ProSet we refer to [Doberkat *et al.*, 1992a]. The high-level structures which ProSet provides qualify the language for prototyping. For a full account to prototyping with set-oriented languages we refer to [Doberkat and Fox, 1989]. A case study for prototyping using SETL is documented in [Kruchten *et al.*, 1984]. The application of SETL for prototyping algorithms for parallelizing compilers is described in [Padua *et al.*, 1993].

4.1 Data Structures

ProSet provides the first-class data types `atom`, `integer`, `real`, `string`, `boolean`, `tuple`, `set`, `function`, `modtype`, and `instance`. It is a *higher-order* language, because functions, modules, and module instances have first-class rights. *First-class* means to be expressible without giving a name. It implies being storable in variables and in data structures, being comparable for equality with other values, and being returnable from or passable to a procedure. Both SETL and ProSet are weakly typed, i.e., the type of an object is in general not known at compile time. Each variable or constant is meant to be an *object* for our terminology. Integer, real, string and Boolean values are used as usual. Atoms are unique with respect to one machine and across machines. They can only be created and compared for equality. Atoms can be created via a call to the standard library function `newat`, which returns a new, unique atom. The unary `type` operator returns a predefined type atom corresponding to the type of its operand.

Atom, integer, real, string, boolean, function, and module values are basic, because they are not built from other values. Tuples and sets are compound data structures, which may be heterogeneous composed of basic values, tuples, and sets. Sets are unordered collections while tuples are ordered. The following expression, for example, creates a tuple consisting of an integer, a string, a Boolean, and a set of two reals:

```
[123, "abc", true, {1.4, 1.5}]
```

Such expressions are called *tuple former*. Sets consisting only of tuples of length two are called maps. There is no genuine data type for maps, because set theory suggests handling them this way. The following statement assigns a set, which is also a map, to the variable `M`:

```
M := { [1,"s1"], [2,"s2"], [3,"s3"] };
```

Now the following equalities hold:

```
domain(M) = {1, 2, 3}
 range(M) = {"s1", "s2", "s3"}
     M(1) = "s1"
     M{1} = {"s1"}
```

Domain and range of a map may be heterogeneous. `M{1}` is the *multi-map selection* for relations.

There is also the undefined value `om` which indicates, for example, selection of an element from an empty set. `om` itself may not be element of a set, but of a tuple. Conceptually a tuple is an infinite vector with almost all components equal to the undefined value `om`. Indexing of tuple components starts with the index `1`.

There is a distinction between the meaning of identifiers on the left and right sides of an assignment. The terms l-value and r-value refer to values that are appropriate on the left and right sides of an assignment, respectively [Aho *et al.*, 1986]. That is, r-values are what we usually think of as *values*, while l-values are *locations*. A tuple of l-values is called *multiple l-value*. A tuple has to be assigned to a multiple l-value: the tuple components are assigned to the individual l-values according to their position in the tuple.

ProSet has *value semantics*. All parameters of procedures are transferred by *copying*, not by *reference*. Differences between parameter-passing to procedures are based primarily on whether an actual parameter may represent an r-value or an l-value:

call by value The default parameter transmission in ProSet is *call by value*, i.e., on procedure invocation, the formal parameter will be initialized with the value of the actual parameter.

call by result With *call by result* the formal parameter obtains `om` as an initial value. On exit from the procedure, the current value of the formal parameter will be copied into the actual parameter.

call by value/result With *call by value/result* the formal parameter is initialized with the actual parameter. When the procedure terminates, the l-value of the actual parameter is determined and the current value of the formal parameter is assigned to the l-value of the actual parameter.

A call-by-value parameter is declared with the keyword `rd`. A call-by-result parameter is declared with the keyword `wr`. A call-by-value/result parameter is declared with the keyword `rw`. Indicating no mode is equivalent to `rd` (the default).

ProSet distinguishes between first-class functions and second-class procedures. Only first-class functions may be assigned to variables. An important difference is that second-class procedures may have side effects on global variables. Global variables are declared with the keyword `visible`. First-class functions cannot have side effects. To convert a second-class procedure into a first-class function one applies the `closure` constructor to the procedure name. When the `closure` constructor is applied to a procedure name, the binding of each global name n used inside the procedure to the actual value v of this object is stored (freezing the values). This binding associates n with v. At the beginning of each invocation of the resulting first-class function this binding is restored in such a way that n behaves like an initialized variable which is declared as `visible` in the resulting function. It is not allowed to apply the `closure` constructor to procedures with write parameters. Consequently, side effects and write parameters are not possible for first-class functions.

4.2 Control Structures

The control structures show that the language has ALGOL as one of its ancestors. There are `if`, `case`, `loop`, `while` and `until` statements as usual and in addition some structures that are custom tailored to the compound data structures. First have a look at expressions forming tuples and sets:

```
T := [1 .. 10];
S := {2*x: x in T | x > 5};     -- result: {12, 14, 16, 18, 20}
```

The iteration "`x in T`" implies a loop in which each element of the tuple `T` is successively assigned to `x`. The visibility of `x` is bound to the set former. For all elements of `T`, which are satisfying the condition "`x > 5`" the result of the expression "`2*x`" is added to the initially empty set. As usual in set theory | means *such that*. With this knowledge the meaning of the following `for` loop should be obvious:

```
for x in S | x > 15 do <statements> end for;
```

The iteration proceeds over a copy, which is created first. The statements are executed for each iteration. The quantifiers (∃, ∀) of predicate calculus are provided, e.g.:

```
if exists x in S | p(x) then <statements> end if;
```

Additionally, ProSet provides the `whilefound` loop:

```
whilefound x in S | p(x) do <statements> end whilefound;
```

The loop body is executed each time an existentially quantified expression with the same iterator would yield `true`. The loop terminates when an existentially quantified expression with the same iterator would yield `false`. The bound variables are local to the `whilefound` loop as they are in `for` loops and in quantified expressions. Unlike `for` loops the iterator is reevaluated for every iteration.

4.3 Introductory Examples

To provide a first impression of ProSet we present two small example programs. In Fig. 4.1 a solution for the so-called *queens' problem* is given. Informally, the problem may be stated as follows:

Is it possible to place n queens ($n \in \mathbb{N}$) on an $n \times n$ chessboard in such a way that they do not attack each other?

Anyone familiar with the basic rules of chess also knows what *attack* means in this context: in order to attack each other, two queens are placed in the same row, the same column, or the same diagonal.

The program in Fig. 4.1 does not solve the above problem directly. It prints out the set of all positions in which the n queens do not attack each other. If it is not possible to place n queens in non-attacking positions, this set will be empty. We denote fields on the chessboard by pairs of natural numbers for convenience (this is unusual in chess, where characters are used to denote the columns). `[1,1]` denotes the lower left corner. The program in Fig. 4.1 with $n = 4$ produces the following set as a result:

```
{{[1,3], [2,1], [4,2], [3,4]},
 {[3,1], [1,2], [2,4], [4,3]}}
```

```
program Queens;
   constant N := 4;
begin
   fields := {[x,y]: x in [1..N], y in [1..N]};

   put ({NextPos: NextPos in npow(N, fields) | NonConflict(NextPos)});

   procedure NonConflict (Position);
   begin
      return forall F1 in Position, F2 in Position |
                   ((F1 /= F2) !implies
                      (F1(1) /= F2(1) and   -- Different columns?
                       F1(2) /= F2(2) and   -- Different rows?
                       abs(F2(1)-F1(1)) /= abs(F2(2)-F1(2))
                                            -- Different diagonals?
                      ));
   end NonConflict;

   procedure implies (a, b);
   begin
      return not a or b;
   end implies;
end Queens;
```

Figure 4.1: The queens' problem.
`npow(k, s)` yields the set of all subsets of the set `s` which contain exactly `k` elements. `NonConflict` checks whether the queens in a given position do not attack each other. It is possible to use procedures with appropriate parameters as user-defined operators by prefixing their names with the "`!`" symbol. This is done here with the procedure `implies`. The predefined function `abs` yields the absolute value of its integer argument. `T(i)` selects the `i`th element from tuple `T`.

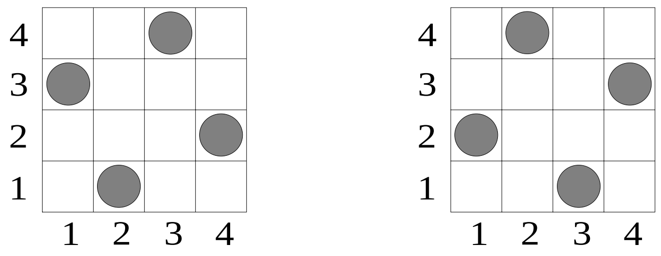


Figure 4.2: The non-attacking positions for $n = 4$ queens.

Since sets are unordered collections, the program may print the elements in different orders. Fig. 4.2 displays the non-attacking positions for $n = 4$ queens. Note that there are no explicit loops and that there is no recursion in the program. All iterations are done implicitly. One may regard this program also as a (executable) specification of the queens' problem.

The program in Fig. 4.3 for topologically sorting the nodes of a directed graph provides a second example. A directed edge between the nodes x and y is indicated by listing the pair `[`x`,`y`]` in the set `edges`. Thus `edges` is a multi-valued map, assigning each node x the set `edges`$\{x\}$ of its successors. The `edges` could also be read in from the terminal or constructed in other ways, instead of using a set former.

The program checks at first if the given graph contains a cycle. If the graph contains a cycle, we cannot compute a topological order. If the graph does not contain a cycle, the nodes are ordered in the initially empty tuple `SortTup`. The nodes without predecessor (the roots) are successively added to `SortTup`. One possible result would be the following tuple in `SortTup`:

```
[1, "a", 2, "b", 3, "c", 4, "d", 5]
```

Fig. 4.4 displays the topological order given by the input set `edges`.

4.4 Exception Handling

An exception is a situation, which, once detected, interrupts the execution of the operation by which it is raised and which subsequently has to be communicated to the caller of that operation. With the notion of *raising an exception* it is merely described that the occurrence of an exception is communicated to the caller and that control is handed over to an exception handler. Exception handlers are similar to procedures, except that they are only invoked by raising exceptions and that they determine the flow of control in a different way. A handler can either terminate the signaling procedure with the `return` statement or resume the signaler with the `resume` statement (return the control to the signaler).

ProSet supports the following forms of raising exceptions by a *signaler*:

`signal`: The signaling procedure allows the handler to either terminate or resume it.

`notify`: The caller obtains a progress report of the evaluation or is requested to carry out some evaluations not implemented by the signaler. This mechanism is akin to coroutines.

The signaler must be resumed by the handler.

`escape`: The signaler must be terminated by the handler.

It is possible to specify a handler for an exception by annotating a statement with a binding between exception name and handler name:

statement `when MyException use MyHandler;`

We refer to [Goodenough, 1975] for a discussion of exception handling in general and to [Doberkat *et al.*, 1992a] for a discussion of exception handling in ProSet.

4.5 Persistence

Persistence of data is characterized by the fact that these data outlive the program that generated them; this is in contrast to volatile data which vanish once the program ceases running. Persistence

```
program TopSort;
begin
   edges := {[1,2], [2,3], [2,4], [3,5], ["a","b"], ["b","c"], ["b","d"]};

   if ContainsCycle (edges) then
      put ("The graph contains a cycle");
   else
      SortTup := [ ];
      nodes := domain(edges) +         -- {1,2,3,"a","b"}
               range(edges);           -- {2,3,4,5,"b","c","d"}
      -- Successively adding remaining roots to SortTup
      -- and remove them from edges and nodes:
      whilefound x in nodes | not (exists y in nodes | (x in edges{y}))
      do
         SortTup with:= x;
         edges lessf:= x;
         nodes less:= x;
      end whilefound;
      put (SortTup);
   end if;

   procedure ContainsCycle (edges);
   begin
      nonleafs := domain (edges);       -- {1,2,3,"a","b"}
      -- Successively remove nonleafs that do not point to nonleafs:
      whilefound x in nonleafs | (edges{x} * nonleafs = { })
      do
         nonleafs less:= x;
      end whilefound;
      return (nonleafs /= { }) ;
   end ContainsCycle;
end TopSort;
```

Figure 4.3: Topologically sorting the nodes of a directed graph.
The binary operators `with` and `less` add elements to and remove elements from sets and tuples, respectively. `lessf` removes all pairs from a map whose first element is the specified one. Set union and intersection are denoted by the binary operators `+` and `*`, respectively.

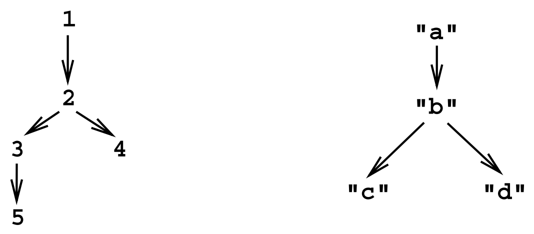

$$\begin{tikzcd}
 & 1 \arrow[d] & & & \text{"a"} \arrow[d] & \\
 & 2 \arrow[dl] \arrow[dr] & & & \text{"b"} \arrow[dl] \arrow[dr] & \\
3 \arrow[d] & & 4 & \text{"c"} & & \text{"d"} \\
5 & & & & &
\end{tikzcd}$$

Figure 4.4: The topological order for the input set
`{[1,2], [2,3], [2,4], [3,5], ["a","b"], ["b","c"], ["b","d"]}` .

is an orthogonal property of values in ProSet. Each value enjoying first-class rights may be made persistent using the name with which it has been defined as a handle. Making use of a persistent value requires indicating this fact in the scope where this is to happen. This scope is implicitly a critical section on the indicated persistent value for invocations of the corresponding program or procedure. Persistence in ProSet is discussed in [Doberkat, 1992] and the relation with ProSet's support for generative communication is discussed in [Doberkat *et al.*, 1993] (see also Sect. 7.15).

4.6 Modules

ProSet makes modules as templates available. Using a module requires instantiating the corresponding template. Instantiation requires providing values for the parameters imported by the module. It has the effect of

- executing the template's initialization code,
- making the exported items available,
- returning an instantiation of the template.

Consider the simple module `IntMod` in Fig. 4.5. The module exports the procedure `increment`. The module has the static variable `i` which is visible in all local procedures and which will be initialized to `0` as specified in the initialization part. This part contains code to be executed exactly once at the time the module is instantiated. The procedure `increment` increments `i`. The module is instantiated as shown in Fig. 4.6 which executes the initialization code and makes the procedure `MyInst.increment` available. The procedures in the `export` list are available after instantiation by prefixing their names with the name of a module instance, provided that they are imported on instantiation.

Modules form scopes of their own, and their visibility follows the scope rules of the language. This is similar to procedures; in particular the following properties are observed:

- names declared as `visible` in an enclosing scope are visible to the module, hence to all local procedures, modules and exception handlers, all other names in an enclosing scope are not accessible to the module,
- names local to the module and visible in the module's body retain their respective bindings across invocations of procedures exported from a module.

The latter property represents the characterizing difference between modules and collections of nested procedures. The `closure` operator has the effect of freezing the values of non-local names visible to the module and making them visible to the module, and it results in a value of type `modtype`.

4.7 Summary

We presented a brief introduction to the basic concepts of ProSet and two introductory examples. The advanced features for exception handling, persistence and modules were sketched. In the remainder of Part II of this thesis we will present and discuss the integration of generative communication into ProSet.

```
module IntMod  export increment;
       visible i;    -- static variable for module instances

       begin
            i := 0;  -- initialization code

       procedure increment ();
       begin
           i := i + 1;
       end increment;
end IntMod;
```

Figure 4.5: A simple Module.

```
        MyInst := instantiate closure IntMod
                        import increment;
                  end instantiate;
```

Figure 4.6: Instantiating a Module.

Chapter 5

Informal Semantics of ProSet-Linda

This chapter presents an informal specification of generative communication in ProSet. The following sections will discuss process creation and termination, the management of multiple tuple spaces, and the tuple-space operations in ProSet. Tuple spaces are regarded primarily as a device for synchronization and communication between processes, and only secondarily for process creation in our approach. Consequently, process creation in ProSet is not a tuple-space operation as it is in C-Linda (see also Sect. 7.2 for a discussion on this subject). However, process creation may be combined with the tuple-space operations as will be sketched in Sect. 5.4.1. Process creation and tuple-space communication are *orthogonal* concepts in ProSet. Orthogonality means that any composition of basic primitives should be allowed [Ghezzi and Jazayeri, 1982].

5.1 Process Creation

In this section we will present an adaptation of the approach for process creation known from Multilisp to set-oriented programming, where new processes may be spawned inside and outside of tuple space. Futures in Multilisp and the concept of touching (see Sect. 3.2.3) provide a method for process creation but no means for synchronization and communication between processes, except for waiting for each other's termination. In our approach the concept for process creation via futures is adapted to set-oriented programming.

Multilisp is based on Scheme [Abelson *et al.*, 1987], which is a dialect of Lisp with lexical scoping. Lisp and Scheme manipulate pointers. This implies touching in a value-requiring context and transmission in a value-ignoring context. This is in contrast to ProSet that uses value semantics, i.e., a value is never transmitted by reference. However, there are a few cases where we can ignore the value of an expression: if the value of an expression is assigned to a variable, we do not need this value immediately, but possibly in the *future*.

Process creation in ProSet is provided through the unary operator `||`, which may be applied to an expression (preferably a function call). A new process will be spawned to compute the value of this expression concurrently with the spawning process analogously to futures in Multilisp.

If this *process creator* `||` is applied to an expression that is assigned to a variable, the spawning process continues execution without waiting for the termination of the newly spawned process. At any time the *value* of this variable is needed, the requesting process will be suspended until the future resolves (the corresponding process terminates) thus allowing concurrency between the *computation* and the *use* of a value. Consider the following statement sequence to see an example:

```
x := || p();        -- Statement 1
...                 -- Some computations without access to x
y := x;             -- Statement 2
```

We assume for simplicity that `p` is a first-class function. If it is a second-class procedure, the `closure` constructor of ProSet must be applied to the procedure name to yield a first-class function (see Sect. 4.1). Consequently, side effects and write parameters are not allowed for parallel processes. Communication and synchronization is done only via tuple-space operations. We shall use the term *procedure* synonymous to *function* where it is not necessary to distinguish.

After statement 1 is executed in the above example, process `p()` runs in parallel with the spawning process. Statement 2 will be suspended until `p()` terminates, because a copy is needed (value semantics). This is in contrast to Lisp where an assignment would copy the address and ignore the value. If `p()` resolves before statement 2 has started execution, then the resulting value will be assigned immediately.

Additionally, if a compound data structure is constructed via a set or tuple forming enumeration, and this data structure is assigned immediately to a variable, we do not need the values of the enumerated components immediately, thus the following statement allows concurrency as above:

```
x := { || p(), 123, || q() };
```

If we replace statement 1 in the previously discussed statement sequence by this statement, then concurrency would be achieved as before. Such parallel set or tuple forming expressions may be compared with constructing lists via the function `cons` in Multilisp, where the list components are also not touched [Halstead, 1985, page 511]. Compound data structures in ProSet are always touched as a whole. Access to tuple or set components as in

```
x := [ || p(), || q() ];
y := x(1);
```

touches the whole tuple or set, thus both, `p()` and `q()`, have to terminate before `x(1)` is accessible. This is in contrast to, for example, I-Structures [Arvind *et al.*, 1989] which provide special kinds of arrays, whose components are evaluated in parallel. With I-Structures, access to array components touches only the requested components and not the array as a whole. It is not really necessary for tuples to be touched as a whole in ProSet, but to handle compound data structures similar to I-Structures would cause problems for selections from sets. We decided to touch compound data structures always as a whole to retain consistency for tuples and sets.

Returning an expression, which is prefixed by `||`, with ProSet's `return` statement achieves concurrency according to the context of the corresponding procedure invocation. Conversely, the actual parameters for procedures as in the procedure call

```
anyproc ( || p(), || q() );
```

are evaluated concurrently to each other, but before `anyproc` is invoked. This is due to the value-passing semantics of procedure invocations in ProSet, where all actual parameters have to be evaluated before the procedure is started (Sect. 4.1).

If the process creator `||` is applied in an expression that is an operand to any operator, then this operator will wait for the return value of the created process. Operators *always* need the values of their operands, and thus have to wait for the termination of processes which compute their operands. For instance, in the following expressions the return values are needed:

```
      1 + || p()
   ["x"] + [|| p()]
```

As any other operator, the process creating operator `||` touches the value of its operand. Hence, an expression such as "`|| || p()`" does not make much sense: the leftmost `||` has to wait for the termination of `p()`. However, it is syntactically correct.

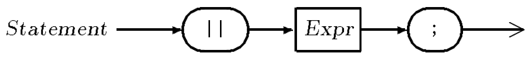
In summary: concurrency is achieved only at creation time of a process and maintained on immediately assigning to a variable, storing in a data structure, returning as a result from a procedure, and depositing in tuple space (this is discussed in Sect. 5.4.1). Every time one tries to obtain a copy one has to wait for the termination of the corresponding process and obtains only then the returned value.

Process-Spawning Statements

Also the following statement, which spawns a new process, is allowed:

```
|| p();
```

The return value of such a process will be discarded. The general form of such a process-spawning statement is as follows:



Note that processes have *no* first-class rights in ProSet (see also Sect. 7.11 for a discussion on this subject). The `type` operator (see Sect. 4.1) has to wait for termination as any other operator.

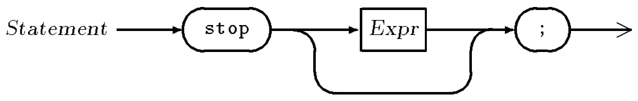
5.2 Program and Process Termination

Analogously to statements, concurrency is achieved in declarations like

```
constant c := || p();
visible x := [ || p() ];
```

If, after such a declaration or a similar statement, `x` is assigned a new value, then the corresponding spawned process will be abandoned provided it is still running. Note that killing processes has to be done with care; especially when such processes are still doing tuple-space operations. Tuple-space operations only have effects when they could be completed.

ProSet's `stop` statement terminates the execution of a process or of an application program:



We distinguish two cases:

1. When executed in a spawned process, this process will be terminated in the same way, as if a `return` statement with the same expression had been executed in the main procedure of this process. If no expression is specified, then `om` will be returned as usual in ProSet.

2. When executed in a main program, which has been started from the operating system, the whole application is terminated and the value of the optional expression is passed to the operating system. Its meaning depends on the operating system. If no expression is specified, a success code is passed to the operating system by default. All spawned processes of this application will be terminated. There exists implicitly a "`stop;`" statement at the end of every main program.

A concurrent program terminates when all its sequential processes have terminated. Termination of the main program terminates the entire application and thus all spawned processes.

5.3 Handling Multiple Tuple Spaces

ProSet supports the use of multiple tuple spaces. Atoms are used to identify tuple spaces. As mentioned in Sect. 4.1, atoms are unique for one machine and across machines. They have first-class rights. ProSet provides four library functions to handle multiple tuple spaces:

`CreateTS(limit)`: Calls ProSet's standard function `newat` to return a fresh atom. The tuple-space manager is informed to create a new tuple space represented/identified by this atom. The atom will be returned by `CreateTS`. Therefore, we can only use atoms that were created by `CreateTS` to identify tuple spaces.

Since one has exclusive access to a freshly assigned tuple-space identity, `CreateTS` provides the basis for information hiding to tuple-space communication.

The integer parameter `limit` specifies a limit on the expected or desired size of the new tuple space. This size limit denotes the total number of passive and active tuples, which are allowed in a tuple space at the same time. `CreateTS(om)` would instead indicate that the expected or wanted size is unlimited regarding user-defined limits, not regarding physical limits. A negative limit is equivalent to 0 (no tuples may be deposited into such a tuple space).

In C-Linda from Scientific Computing Associates the size for the global tuple space is specified in byte blocks [Sci, 1992]. The level of such a unit is too low for a prototyping language.

`ExistsTS(TS)`: Yields `true`, if `TS` is an atom that identifies an existing tuple space; else `false`.

`ClearTS(TS)`: Removes all active and passive tuples from the specified tuple space. This operation is executed indivisible for the specified tuple space.

This function appears to be useful, for instance, in a master-worker application: when the work has been done, the master can remove garbage and abandon the workers. See also the examples in Chap. 6 for applications of `ClearTS`.

`RemoveTS(TS)`: Calls `ClearTS(TS)` and removes the specified tuple space from the list of existing tuple spaces.

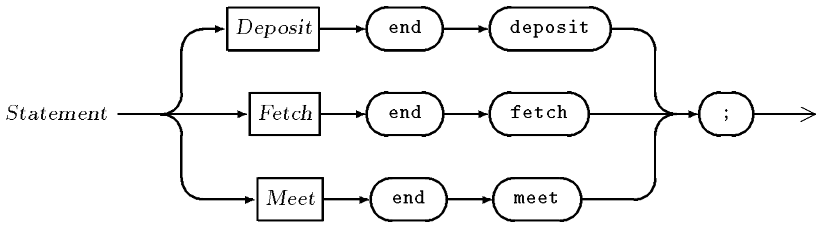
If the functions `ExistsTS`, `ClearTS`, or `RemoveTS` are invoked with actual parameters which are not atoms, or if the function `CreateTS` is invoked with an actual parameter which is not an integer or `om`, the predefined exception `type_mismatch` will be raised. If the functions `ExistsTS`, `ClearTS`, or `RemoveTS` are called with an atom, which is not a valid tuple-space identity, then the predefined exception `ts_invalid_id` will be raised.

By way of introduction, a first-class tuple-space identity is a *coordination capability* which may be bound to variables, passed as an argument (or returned as a result from) procedures, or embedded within compound data structures.

Every ProSet program has its own tuple-space manager. Tuple spaces are not persistent. They exist only until all processes of an application have terminated their execution. Consequently, tuple space communication in ProSet as presented here is designed for *multiprocessing* (single application running on multiple processors) as opposed to *multiprogramming* (separate applications). See also Sect. 7.15 for a discussion on this subject.

5.4 Tuple-Space Operations

ProSet provides three tuple-space operations for communication and synchronization of parallel processes:



The `deposit` operation deposits a new tuple into a tuple space, the `fetch` operation fetches and removes a tuple from a tuple space, and the `meet` operation *meets* and leaves a tuple in a tuple space. It is possible to change the tuple's value while meeting it. A concise overview of the abstract grammar for the tuple-space operations is presented in Appendix A using BNF (Backus Naur Form).

There is no difference between ProSet-tuples and passive Linda-tuples. Passive Linda-tuples contain only evaluated values and, therefore, no active processes (see also Sect. 3.2.9). Linda and ProSet both provide tuples thus it is quite natural to combine them on the basis of this common feature. However, a tuple space is a multiset of tuples, whereas the type system of ProSet does not directly provide the notion of multisets or bags. One could model multisets, for example, via maps from tuples to counts, but this would not reflect the matching provided by tuple spaces. We will discuss the introduction of a type "*tuple space*" in Sect. 7.12.

5.4.1 Depositing Tuples

The `deposit` operation deposits a tuple into a specified tuple space. We distinguish between passive and active tuples in tuple space. If there are no executing processes in a tuple, then this tuple is added as a passive one:

```
deposit [ 123, "mystring", 3.14 ] at TS end deposit;
```

This statement deposits the tuple `[ 123, "mystring", 3.14 ]`. `TS` is the tuple space at which the specified tuple has to be deposited. The expression after the keyword `at` within tuple-space operations has to yield a valid tuple-space identity. If not, the exception `ts_invalid_id` will be raised. Note that the exception `type_mismatch` and not `ts_invalid_id` will be raised if an expression after the keyword `at` within tuple-space operations yields *not* an atom.

If processes are spawned in a tuple, then this tuple is added as an active one to the tuple space:

```
deposit [ "myprocess", || p() ] at TS end deposit;
```

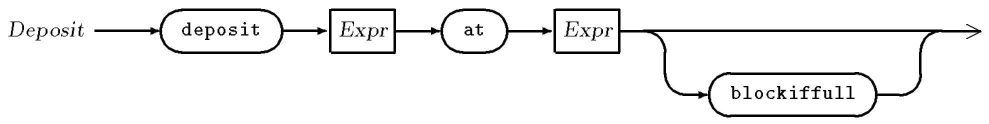
Depositing a tuple into a tuple space does not touch the value (see also Sect. 5.1). When all processes in an active tuple have terminated their execution, then this tuple converts into a passive one with the return values of these processes in the corresponding tuple fields. Active tuples are invisible to the other tuple-space operations until they convert into passive tuples. The other two tuple-space operations apply only to passive tuples.

Note that the following statement sequence deposits a *passive* tuple:

```
x := [ "myprocess", || p() ];
deposit x at TS end deposit;
```

The `deposit` operation copies the value of `x`, and thus has to wait for the termination of `p()` (see also Sect. 5.1).

The syntax for the `deposit` operation is as follows:



As indicated above, the first expression must yield a tuple to be deposited at the tuple space and the second expression must yield a valid tuple-space identity. If not, the exception `type_mismatch` or `ts_invalid_id` will be raised.

Here, a rather aesthetical question arises. The syntax for ProSet's tuple-space operations contains some redundancies. A `deposit` operation such as

```
deposit [...] at TS end deposit ;
```

could simply be written without the tail:

```
deposit [...] at TS ;
```

Conversely, all complex constructs of ProSet are enclosed within head/tail pairs like

```
anyconstruct ...  end anyconstruct
```

To retain consistency, we include the `end`-tails in the tuple-space operations.

Limited Tuple Spaces

Because every existing computing system has only finite memory, memory for tuple spaces will also be limited. Usually, tuple-space communication does not deal with *full* tuple spaces: ideally, there is always enough room available. Thus most runtime systems for Linda hide the fact of limited memory from the programmer (see also Sect. 7.6 for a discussion on this subject).

In ProSet, the predefined exception `ts_is_full` will be raised by default if no memory is available for a `deposit` operation. This exception is raised with ProSet's `signal` statement. Signal exceptions permit the function raising the exception to be either terminated or resumed at the handler's discretion (see also Sect. 4.4). It is possible to specify a handler for an exception by annotating a statement with a new binding between exception name and handler name:

```
deposit [ x ]
     at TS
end deposit when ts_is_full use MyHandler;
```

If the associated handler then executes a `return` statement, the statement following the `deposit` will be executed and the tuple of the respective `deposit` will not be deposited. If the handler executes a `resume` statement, the `deposit` operation tries again to deposit the tuple. See Sect. 6.1 for an example.

The exceptions `ts_invalid_id` and `type_mismatch` are always raised with ProSet's `escape` statement, which prohibits resuming (see Sect. 4.4).

Optionally, the programmer may specify that a `deposit` operation will be suspended on a full tuple space until space is available again:

```
deposit [ x ] at TS
        blockiffull
end deposit;
```

A new binding between the predefined exception `ts_is_full` and another handler name should not be specified together with `blockiffull`, because `ts_is_full` would never be raised when `blockiffull` is specified. The compiler rejects this.

The suitable handling of full tuple spaces depends on the application one has to program. Thus a general setting does not seem to be appropriate. Blocking is useful, for example, in a producer-consumer application. In a producer-consumer application a producer produces data for a consumer [Andrews, 1991]. The producer would wait when the consumer cannot consume the data fast enough. In a master-worker application (see Sect. 6.1) one might prefer to collect some results by an own handler before producing more tasks, when the tuple space is full. In Sects. 6.1 and 6.2 master-worker applications with limited tuple spaces will be presented.

5.4.2 Fetching Tuples

A `fetch` operation tries to fetch and remove exactly one tuple from a tuple space. It is possible to specify several templates for the specified tuple space in a statement, but only one template may be selected nondeterministically (see also Sect. 5.4.4). We start with a first example for a `fetch` operation with a single template:

```
fetch ( "name", ? x |(type $(2) = integer) ) at TS end fetch;
```

This template only matches tuples with the string `"name"` in the first field and integer values in the second field. The symbol `$` may be used like an expression as a placeholder for the values of corresponding tuples in tuple space. The expression `$(i)` then selects the $\mathtt{i}^{th}$ element from these tuples. Indexing starts with `1`. It is only allowed to use the symbol `$` this way in expressions that are parts of templates. As usual in ProSet, `|` means *such that*. The Boolean expression after `|` may be used to *customize* matching by restricting the set of possibly matching tuples. ProSet employs *conditional value matching* and not type matching known from C-Linda and similar embeddings of Linda into statically typed languages. A tuple and a template match iff all the following conditions hold:

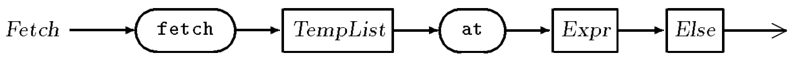
- The tuple is passive.
- The arities are equal. The arity of a tuple or template is the largest index of a tuple or template component which is not the undefined value `om`. As usual, indexing starts with `1`.
- Values of actuals in templates are equal to the corresponding tuple fields.
- The Boolean expression after `|` in the template evaluates to `true`. If no such expression is specified, then `true` is the default.

The l-values (Sect. 4.1) specified in the formals (the variable `x` in our example) are assigned the values of the corresponding tuple fields provided matching succeeds. The selected tuple is removed from tuple space. If no `else` statements are specified as in the above example then the statement suspends until a match occurs. If statements are specified for the selected template, these statements are executed. An example with multiple templates, associated statements, and an `else` statement follows:

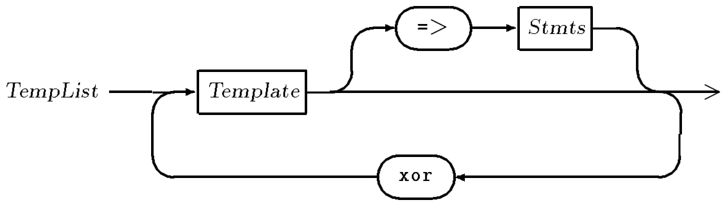
```
fetch ( "name", ? x |(type $(2) = integer) ) => put("Integer fetched");
  xor ( "name", ? x |(type $(2) = set) )     => put("Set fetched");
    at TS
 else put("Nothing fetched");
end fetch;
```

Here both templates consist of an *actual* (the expression `"name"`), a *formal* preceded by a question mark, and a template condition. The templates are enclosed in parentheses and not in brackets in order to set the templates apart from tuples. The `else` statement will be executed if none of the templates matches.

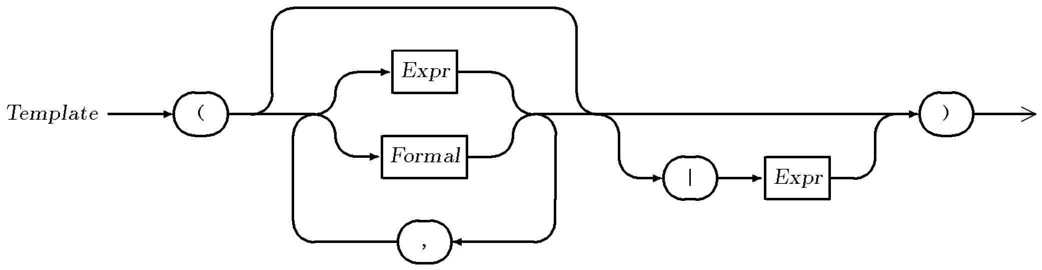
The syntax for the `fetch` operation is as follows:



The expression must yield a valid tuple-space identity as before. The syntax for the template list is as follows:

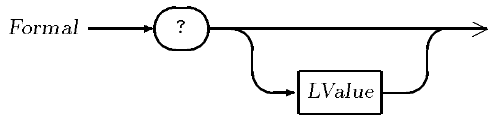


We use the keyword `xor` (exclusive or) and not `or` to separate the individual alternatives to emphasize that only one template may be selected. The optional statements are preceded by the symbol =>, because the same syntax was chosen for ProSet's `case` statement [Doberkat *et al.*, 1992a]. A template then consists of a list of ordinary expressions (the actuals), formals, and a template condition (after |):



The Boolean expression after |, which may be used to customize matching, is placed within the parentheses of the templates to make the implicit scope of $ explicit. These Boolean expressions have to be side-effect free, because it depends on the implementation how often such a condition is evaluated while the tuple-space manager tries to match tuples and templates. Note that the ProSet-Compiler can check at compile-time if a procedure has side effects or not.

The *actuals* are the expressions in the template list. They are first evaluated in arbitrary order. Note that a template may be empty to match the empty tuple `[]`. The fields that are preceded by a question mark are the *formals* of the template:

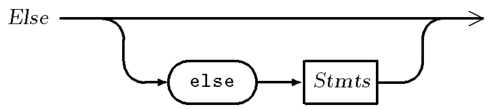


The *l*-values specified in the formals are assigned the values of the corresponding tuple fields provided matching succeeds. If an *l*-value is specified more than once, it is not determined which one of the possible values is assigned. If an *l*-values also occurs in the expressions of templates, then these expressions are evaluated with the corresponding *old* values of these *l*-values. If no *l*-value is specified, then the corresponding value will not be available. A formal without an *l*-value may be regarded as a "don't care" or "only take care of the condition" field.

Note that the template `(1,om)` matches the tuple `[1]`. The equation "`[1]=[1,om]`" holds in ProSet (see Sect. 4.1). Note also that the template `(om,1)` does *not* match the tuple `[1]`.

Non-blocking Matching

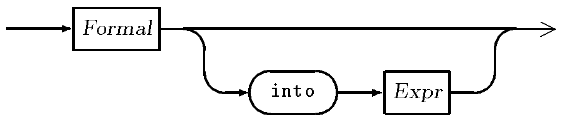
It is possible to specify `else` statements to be executed, if none of the templates matches:



We will use the notion *non-blocking matching* if `else` statements are specified as opposed to *blocking matching* if no `else` statements are specified. In Sect. 7.6 a discussion on this subject can be found.

5.4.3 Meeting Tuples

The `meet` operation tries to *meet* and leave exactly one tuple in a tuple space. It is possible to change the tuple while meeting it. Except for the fact that a `meet` operation, which does not change the tuple met, leaves the tuple it found in tuple space, it works like the `fetch` operation. Exchanging the keyword `fetch` with `meet` and the nonterminal $\boxed{\textit{Formal}}$ with



in the syntax diagrams of Sect. 5.4.2, one obtains the syntax for the `meet` operation. The complete abstract grammar for the tuple-space operations is presented in Appendix A. An example for the `meet` operation follows:

```
meet  ( "name", ? x |(type $(2) = integer) ) => put("Integer met");
  xor ( "name", ? x |(type $(2) = set) )     => put("Set met");
   at TS
 else put("Nothing met");
end meet;
```

The expressions are evaluated as usual, the formals are used to create templates, which are used for matching as with the `fetch` operation. If no `else` case is specified, then the statement suspends until a match occurs. The values of the tuple fields that were fetched for the corresponding formals of the template are assigned to the corresponding *l*-values. If statements are specified for the selected template, these statements are executed (only for this template).

If there are no `into`s specified as in this example, then the selected tuple is *not* removed from tuple space. This case may be compared with the `rd/rdp` operations of C-Linda (see Sect. 3.2.9). Except for the fact that the `meet` operation without `into`s leaves the tuple it found in tuple space, the construction works like the `fetch` operation.

Changing Tuples

The absence of support for user-defined high-level operations on shared data in Linda is criticized [Bal, 1992]. We agree that this is a shortcoming. For overcoming it we allow to change tuples while meeting them in tuple space. This is done by specifying expressions `into` which specific tuple fields will be changed. Tuples, which are met in tuple space, may be regarded as shared data since they remain in tuple space irrespective of changing them or not.

If there are `into`s specified, then the tuple is at first fetched from the tuple space as it would be done with the `fetch` operation. Afterwards a tuple will be deposited into the same tuple space, where all the tuple fields without `into`s are unchanged and all the tuple fields with `into`s are updated with the values of the respective expressions. Consider

```
meet ( "name", ? into $(2)+1 ) at TS end meet;
```

which is equivalent to the series of statements with `x` as a fresh name:

```
fetch ( "name", ? x ) at TS end fetch;
deposit [ "name", x+1 ] at TS end deposit;
```

Indivisibility is guaranteed, because fetching the passive tuple at starting and depositing the new passive or active one at the end of the user-defined operation on shared data are atomic operations. Note that the tuple is not really removed from the tuple space. The above equivalence is only introduced to specify the semantics, not the implementation.

Therefore, with the `meet` operation expensive copying of compound data may be avoided. Consequently, the `meet` operation will not raise `ts_is_full` when a tuple space is full while depositing a changed tuple, when the number of allowed tuples in this tuple space is exceeded (see also Sect. 5.3). The place for the met tuple will be reserved for the entire operation. However, if the changed tuple exceeds a physical memory limit, this will raise `ts_is_full`.

It is not necessary to specify *l*-values for changing tuple fields. However, they may be used:

```
meet ( "name", ? x into $(2)+1 ) at TS end meet;
```

Here `x` is assigned the value of the corresponding tuple field *before* the change takes place. Remember that it is only allowed to use the symbol `$` this way in expressions that are parts of templates.

5.4.4 Nondeterminism and Fairness while Matching

There are two sources for nondeterminism while matching:

1. Several matching tuples exist for a given template: one tuple will be selected nondeterministically.
2. A tuple matches several templates: one template will be selected nondeterministically.

If in any case there is only one candidate available, this one will be selected. There are several ways for handling fairness while selecting tuples or templates that match if there are multiple candidates available. We assume a fair scheduler to guarantee process fairness, which means that no single process is excluded of CPU time forever. We will now discuss *fairness of choice* which is important for handling the nondeterminism derived from matching. There exist some fairness notions [Kwiatkowska, 1989]:

Unconditional Fairness Every process will be selected infinitely often.

Weak Fairness If a process is enabled continuously from some point onwards then it eventually will be selected. Weak Fairness is also called *justice*.

Strong Fairness If a process is enabled infinitely often then it will be selected infinitely often.

Since unconditional fairness applies only to nonterminating processes we do not consider it for our approach. In ProSet the following fairness guarantees are given for the two sources for nondeterminism as mentioned above:

1. Tuples will be selected without any consideration of fairness.
2. Templates will be selected in a weakly fair way.

Fairness is also important for processes which are blocked on full tuple spaces:

3. Processes which are blocked on full tuple spaces are selected in a weakly fair way when tuples are fetched from the respective tuple spaces.

In cases (2.) and (3.) processes are involved and enabled after selection, whereas in case (1.) this is not the case for deposited tuples. Fairness is a liveness property [Ben-Ari, 1990; Andrews, 1991]. Liveness properties specify that something good will happen (e.g., termination). Our fairness properties specify that every blocked process which may be selected infinitely often is selected eventually. We could also select deposited tuples in a fair way, but this would not increase the liveness of ProSet programs. Therefore, it is reasonable to employ weakly fair selection in cases (2.) and (3.), and unfair selection in case (1.). These fairness properties are specified formally by means of temporal logic in Sect. 8.8 and discussed in Sect. 7.7.

Weakly fair selection of templates applies only to blocking matching: if a template, which is used for non-blocking matching, does match immediately then this one is excluded of further matching and the corresponding process is informed of this fact. This applies accordingly to non-blocking matching with multiple templates, too. Templates (respectively processes), which are suspended because no tuple matches them are weakly fair matched with tuples later deposited. The implementation has to guarantee this.

5.5 Summary

We presented the integration of generative communication into ProSet informally by means of simple examples, syntax diagrams, and explanatory text. We will extend this presentation by means of some example programs in Chap. 6, and continue with a discussion of other proposed approaches to extending Linda and some design alternatives to our approach in Chap. 7.

Chapter 6

Example Programs

In Chap. 5 we present the informal semantics of generative communication in ProSet. Now we illustrate this presentation by means of some example programs. To be more precise, we shall present the parallel solutions for a master-worker application with limited tuple spaces, the queens' problem, matrix multiplication, the traveling salesman problem, and the dining philosophers problem. Our intention is to present the features of ProSet-Linda.

6.1 A Master-Worker Application with Limited Tuple Spaces

A master-worker application with limited tuple spaces, where the master makes *some* new tasks dependent on *some* results produced by the workers will now be presented to discuss the semantics of non-blocking matching in correlation with limited tuple spaces. See also Sect. 7.6 for a discussion of this relation.

In a master-worker application, the task to be solved is partitioned into independent subtasks. These subtasks are placed into the tuple space, and each process in a pool of identical workers then repeatedly retrieves a subtask description from the tuple space, solves it, and deposits the solutions into the tuple space. The master process then collects the results. An advantage of this programming approach is easy load balancing because the number of workers is usually variable.

The program in Fig. 6.1 consists of a master (the main program), a worker procedure, and an exception handler. The master spawns `NumWorker` worker processes outside the tuple space. This number is an argument to the main program. These workers execute in an infinite loop, in which tasks are fetched from tuple space `WORK`, and results are computed and deposited at tuple space `RESULT`. A worker is suspended if there is no more space available for depositing results (`blockiffull`), since a worker does not know what to do in this case. The master repeatedly deposits tasks into tuple space `WORK`. If there is no more space available for new tasks, the exception `ts_is_full` will be raised by the `deposit` operation and the handler `fetchResults` will be activated to fetch some results (see also Sect. 4.4). At least one result will be fetched by the exception handler and probably some more, before resuming the `deposit` operation. The master becomes temporary a worker while executing the handler. It is not important *how many* results are fetched. It is only important that *some* results are fetched to enable the workers.

The program could be made more efficient if only one tuple space would be used, since `fetchResults` may be called earlier and thus the workers are suspended for a shorter time. However, we use two tuple spaces to simplify matching. The master could do something with the results if he wants. He also would have to fetch remaining results when the `whilefound` loop terminates. Notice that the handler `fetchResults` changes the visible set `tasks`. This is not allowed for the spawned workers.

This program presents only a template for such a master-worker application. It is not a complete program. Section 6.2 will present a specific instance with some minor modifications.

```
program MasterWorker;
  constant NumWorker := argv(2); -- program argument
  visible constant WORK := CreateTS (1000),
                   RESULT := CreateTS (500);
  visible tasks := { some task tuples };
begin
    for i in [1..NumWorker] do
        || closure Worker ();  -- start the workers outside of tuple space
    end for;

    whilefound next in tasks do
        deposit next
             at WORK
        end deposit when ts_is_full use fetchResults;
    end whilefound;
    -- fetch remaining results ...

    procedure Worker ();
    begin
       loop
           fetch ( ? MyTask ) at WORK end fetch;
           -- compute MyResult ...
           deposit [ MyResult ] at RESULT
                   blockiffull
           end deposit;
       end loop;
    end Worker;

    handler fetchResults ();
    begin
        -- fetch at least one result:
        fetch ( ? firstResult ) at RESULT end fetch;
        -- fetch some more results:
        loop
            fetch ( ? nextResult ) at RESULT
                  else tasks +:= { some new task tuples };
                       resume;  -- the deposit operation
            end fetch;
            -- do something with the result ...
        end loop;
    end fetchResults;
end MasterWorker;
```

Figure 6.1: A master-worker program with limited tuple spaces.
The keyword `visible` declares the respective objects as visible for the local procedure and the handler. The default scope rules of ProSet declare an object as *hidden* for local procedures. Note that the `closure` constructor has to be applied here to convert the second-class procedure `Worker` into a first-class function (see Sects. 4.1 and 5.1). However, the resulting first-class function will be identical to the original procedure, because `Worker` has no side effects on global variables.

We consider non-blocking matching useful to handle limited tuple spaces, because we do not know the exact number of results we had to fetch in this example (see the `else` statements in `fetchResults`). We argue that the semantic problems with supporting the notion of limited tuple spaces and non-blocking matching are related. See also Sect. 7.6 for a discussion on this subject.

6.2 The Queens' Problem Revisited

In Sect. 4.3 the queens' problem was introduced and a sequential solution presented. In Fig. 6.2 a parallel solution based on the master-worker model, which has been introduced in Sect. 6.1, is given.

The resulting set of non-conflicting positions is built up in tuple space `RESULT` via changing `meet` operations. The master program spawns `NumWorker` worker processes. One could pass this number as an argument to the main program according to the available processors. The counter in tuple space `RESULT` is necessary to let the master wait until all positions are evaluated. Tuple space `WORK` is cleared after work has been done thus also terminating the workers. It would be more efficient to deposit the positions directly into the tuple space instead of first constructing the set `Positions`.

The specification of a limit for the tuple spaces is primary intended here to obtain a facility for load balancing between producers and consumers of tuples (not necessary load balancing between hardware processors, but between processes).

6.3 Parallel Matrix Multiplication

The program in Fig. 6.3 on page 67 presents parallel matrix multiplication. Since ProSet does not provide arrays, we use tuples of tuples to represent two-dimensional matrices. The tuple space `WORK` is used to deposit the rows of matrix `A`, the columns of `B`, and the workers. The result matrix `C` is built up in tuple space `RESULT` via changing `meet` operations. The counter attached to the result matrix is necessary to let the master wait until all elements are computed. Tuple space `WORK` is cleared after work has been done thus also terminating the workers

This program does not support load balancing, because the number of workers is fixed as the number of elements in the result matrix (the number of rows in `A` multiplied by the number of columns in `B`). One could reduce the number of workers as it has been done in [Ahuja *et al.*, 1986, page 30].

6.4 The Traveling Salesman Problem

In this section we consider the traveling salesman problem in which it is desired to find the shortest route that visits each of a given set of cities exactly once. We want to compute an optimal route for some cities in the Ruhrgebiet. The selected cities with their connections and distances are displayed in Fig. 6.4 on page 68. The salesman should start in Essen.

The problem can be solved using *branch-and-bound* [Lawler and Wood, 1966]. It uses a tree to structure the search space of possible solutions. The root of the tree is the city in which the salesman should start. Each path from the root to a node represents a partial tour for the salesman. Leaf nodes represent either partial tours without connections to not yet visited cities or complete tours. Complete tours visit each city exactly once. Fig. 6.5 on page 68 displays the search tree for our selection of cities. The complete tours, in which each city is visited, are set off through thick lines. In general, it is not necessary to search the entire tree: a *bounding rule* avoids searching the entire tree. For the traveling salesman problem, the bounding rule is simple. If the length of a partial tour exceeds the length of an already known complete tour, the partial tour will never lead to a solution better than what is already known.

Parallelism in a branch-and-bound algorithm is obtained by searching the tree in parallel. Our program in Fig. 6.6 on page 69 stores the cities with their connections in the global set `DistTable`. This set

```
program ParallelQueens;
  constant N := 8, NumWorker := argv(2); -- program argument
  visible constant WORK := CreateTS (1000),
                   RESULT := CreateTS (500);
begin
   Positions := npow(N, {[x,y]: x in [1..N], y in [1..N]});
   for i in  [1 .. NumWorker] do          -- spawn the worker processes
      deposit [ || closure Worker() ] at WORK end deposit;
   end for;
   deposit [ {} ] at RESULT end deposit;  -- initialize the result set
   deposit [ 0 ] at RESULT end deposit;   -- initialize the position counter
   for NextPosition in Positions do
      deposit [ NextPosition ]
           at WORK
      end deposit when ts_is_full use DoSomeWork;
   end for;
   fetch ( #Positions ) at RESULT end fetch;  -- wait for the work to be done
   fetch ( ? ResultPos |(type $(1) = set) ) at RESULT end fetch;
   put (ResultPos);
   ClearTS(WORK);

   procedure Worker (); begin
      loop
         fetch ( ? MyPosition |(type $(1) = set) ) at WORK end fetch;
         EvalPosition (MyPosition);
      end loop;
   end Worker;

   handler DoSomeWork (); begin
      fetch ( ? MyPosition |(type $(1) = set) ) at WORK end fetch;
      EvalPosition (MyPosition);
      loop
         fetch ( ? MyPosition |(type $(1) = set) ) at WORK
           else resume;  -- the deposit operation
         end fetch;
         EvalPosition (MyPosition);
      end loop;
   end DoSomeWork;

   procedure EvalPosition (Position); begin
      if NonConflict (Position) then
          meet ( ? into ($(1) with Position) |(type $(1) = set) )
             at RESULT
          end meet;
      end if;
      meet ( ? into ($(1) + 1) |(type $(1) = integer) )
         at RESULT
      end meet;
   end EvalPosition;
end ParallelQueens;
```

Figure 6.2: Parallel solution for the queens' problem.
See Fig. 4.1 on page 46 for the procedure `NonConflict`. The master program spawns `NumWorker` worker processes. This number is an argument to the main program. The unary operator `#` returns the number of elements in a compound data structure. When there is no more space available for more tasks, the exception `ts_is_full` will be raised and the handler `DoSomeWork` will be activated to do some work: the master becomes temporary a worker.

```
program matrix;
   visible constant WORK := CreateTS(om),
                    RESULT := CreateTS(om);
begin
   A := [ [1, 2, 3],
          [4, 5, 6]
        ];
   B := [ [ 7,  8,  9, 10],
          [11, 12, 13, 14],
          [15, 16, 17, 18]
        ];
   C := [ [], [] ];

   for i in [ 1 .. #A ] do
      deposit [ "A", i, A(i) ] at WORK end deposit;
   end for;
   for i in [ 1 .. #B(1) ] do
      ithColumn := [ B(j)(i): j in [1 .. #B] ];
      deposit [ "B", i, ithColumn ] at WORK end deposit;
   end for;
   deposit [ "C", C, 0 ] at RESULT end deposit;
   for i in [ 1 .. #A ], j in [ 1 .. #B(1) ] do
      deposit [ || closure worker(i,j) ] at WORK end deposit;
   end for;
   fetch ( "C", ? C, #A * #B(1) ) at RESULT end fetch;
   ClearTS (WORK);

   procedure worker (i, j);
   begin
      meet ( "A", i, ? row ) at WORK end meet;
      meet ( "B", j, ? column ) at WORK end meet;
      Dot := DotProduct (row, column);
      meet ( "C", ? into AddDot($(2),i,j,Dot), ? into $(3)+1 )
          at RESULT
      end meet;

      procedure DotProduct (row, column);
      begin
         sum := 0;
         for i in [1..#row] do
            sum +:= row(i)*column(i);
         end for;
         return sum;
      end DotProduct;

      procedure AddDot(C, i, j, Dot);
      begin
         C(i)(j) := Dot;
         return C;
      end AddDot;
   end worker;
end matrix;
```

Figure 6.3: Parallel matrix multiplication.

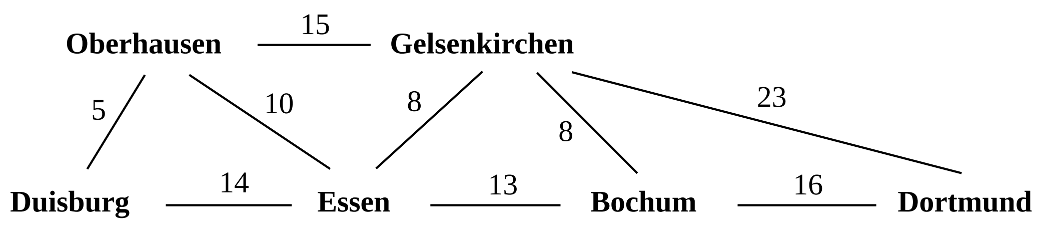


Figure 6.4: Some cities in the Ruhrgebiet with their connections and distances.

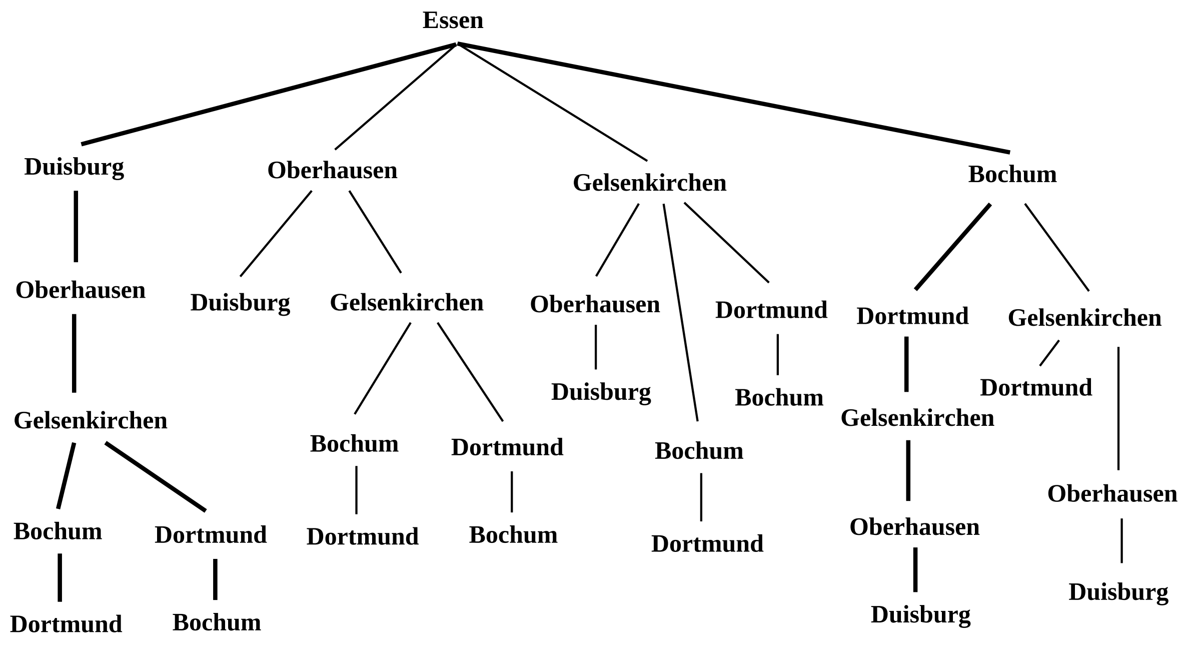


Figure 6.5: The search tree for our selection of cities in the Ruhrgebiet.

```
program tsp;
  visible constant DistTable :=
            {[["Duisburg","Essen"], 14], [["Essen","Duisburg"], 14],
             [["Duisburg","Oberhausen"], 5], [["Oberhausen","Duisburg"], 5],
             [["Oberhausen","Essen"], 10], [["Essen","Oberhausen"], 10],
             [["Oberhausen","Gelsenkirchen"], 15], [["Oberhausen","Gelsenkirchen"], 15],
             [["Essen","Gelsenkirchen"], 8], [["Essen","Gelsenkirchen"], 8],
             [["Essen","Bochum"], 13], [["Bochum","Essen"], 13],
             [["Bochum","Dortmund"], 16], [["Dortmund","Bochum"], 16],
             [["Bochum","Gelsenkirchen"], 8], [["Gelsenkirchen","Bochum"], 8],
             [["Dortmund","Gelsenkirchen"], 23], [["Gelsenkirchen","Dortmund"], 23]},
        Nodes := {"Duisburg", "Essen", "Bochum", "Dortmund", "Oberhausen", "Gelsenkirchen"},
        Minimum := CreateTS (om);
begin
  Start := "Essen";
  deposit [ om, []] at Minimum end deposit; -- The minimum is not yet defined (om)

  NumWorker := 0;  -- Number of spawned workers
  for Entry in DistTable | Entry(1)(1) = Start do
    -- Spawn a worker for each initial route:
    deposit [ || closure Worker (Entry(1), Entry(2))] at Minimum end deposit;
    NumWorker +:= 1;
  end for;
  for i in [1..NumWorker]  do
    fetch ( ? ) at Minimum end fetch; -- wait for the workers
  end for;

  fetch ( ? distance, ? route ) at Minimum end fetch;
  put("Tour de Ruhr = ", route);
  put("Distance = ", distance);

  procedure Worker (MyRoute, MyDistance);
  begin
    meet ( ? distance, ? ) at Minimum end meet;  -- to check if we can return
    if distance /= om and distance <= MyDistance then
      return;  -- there exists already a shorter or equal long route: we prune this subtree
    end if;

    if #MyRoute = #Nodes then
      -- We have a complete route. Change the minimum to our route if it is
      -- still the shortest one:
      meet ( ? into MyDistance, ? into MyRoute
             | ($(1) = om or $(1) > MyDistance ) )
        at Minimum
      end meet;
    else
      -- Call a worker for each route which is a connection of MyRoute with a
      -- node that is not in MyRoute:
      for Entry in DistTable | (Entry(1)(1) = MyRoute(#MyRoute) and
                                Entry(1)(2) notin {x: x in MyRoute}) do
        Worker (MyRoute with Entry(1)(2), MyDistance + Entry(2));
      end for;
    end if;
  end Worker;
end tsp;
```

Figure 6.6: Solution for the traveling salesman problem.

is a map which maps pairs of cities to their distance. The distances are specified for each direction. The distances between two cities may be different for different directions (for one-way connections). The set `Nodes` contains the cities involved. The string `Start` indicates the starting point. We deposit the *current* minimal distance together with the corresponding route in the tuple space `Minimum`. This minimal distance is initially undefined (`om`) and the corresponding route is an empty tuple. The main program spawns for each initial route a worker process to compute the search tree in parallel. After spawning the workers, the main program waits until all workers have done their work, and then the main program fetches the optimal distance together with the corresponding route.

Each worker first checks whether its partial route (the parameter `MyRoute`) exceeds the length of an already known complete route. Then the worker terminates (according to the bounding rule). If the length of the partial tour does not exceed the length of an already known complete route, the worker checks if its partial route is already a complete route. If the partial route is already a complete route, the worker changes the minimal route in the tuple space `Minimum` to the given route, provided that the given route is still the shortest one. If the partial route is not a complete route, the worker calls itself recursively for each route which is a connection of the given route with a node that is not in the given route. There has to be a connection defined in `DistTable` between the last node in the given route and the next node that is not in the given route to constitute a new extended route.

The program in Fig. 6.6 on page 69 prints out:

```
Tour de Ruhr = ["Essen", "Duisburg", "Oberhausen", "Gelsenkirchen", "Bochum", "Dortmund"]
Distance = 58
```

For simplicity we assume that there exists at least one complete route that visits each of a given set of cities exactly once. If such a complete route does not exist, the program prints out:

```
Tour de Ruhr = []
Distance = om
```

Often it is assumed in solutions for the traveling salesman problem that there exists a connection between each pair of cities. Our program does not have this assumption, and also solves problems where the distances between two cities depend on the direction.

6.5 The Dining Philosophers Problem

We conclude this chapter with a classical problem in parallel programming. The dining philosophers problem was posed originally by Dijkstra [Dijkstra, 1971], and is often used to test the expressivity of new parallel languages. We quote for a description of this problem:

> "The 'dining philosophers' problem goes like this: a round table is set with some number of plates (traditionally five); there's a single chopstick between each two plates, and a bowl of rice in the center of the table. Philosophers think, then enter the room, eat, leave the room and repeat the cycle. A philosopher can't eat without two chopsticks in hand; the two he needs are the ones to the left and the right of the plate at which he is seated." [Carriero and Gelernter, 1990a, page 182]

The original problem specification uses a bowl of spaghetti and the utensils were forks [Dijkstra, 1971], but it does not seem immediately reasonable why two forks are needed to eat spaghetti thus we use the above specification. The dining philosophers problem is a popular instance of selective mutual exclusion problems, where parallel processes compete for common resources. Fig. 6.7 presents the general form of a solution for the mutual exclusion problem (see also [Ben-Ari, 1990; Andrews, 1991]).

Some extensions to the dining philosophers problem were proposed:

```
loop
    Non Critical Section ;
    Pre Control ;
    Critical Section ;
    Post Control ;
end loop;
```

Figure 6.7: Form of mutual exclusion solution.

- The drinking philosophers problem, where neighbors are allowed to drink simultaneously provided that they are drinking from different bottles [Chandy and Misra, 1984].
- The evolving philosophers problem with *change management*, where a philosopher can leave the table, a new philosopher can join the table, or a philosopher can move from one part of the table to another [Hazelhurst, 1991].
- The problem of the restaurant for dining philosophers with multiple tables and change management [Ciancarini, 1991].

In this paper we consider the *plain* dining philosophers problem. The solution in ProSet in Fig. 6.8 is derived from the C-Linda version in [Carriero and Gelernter, 1990a]. In the C-Linda version the philosophers first fetch their left and then their right chopsticks. In our ProSet-Linda version this order is not specified. The program works for arbitrary `n > 1`. To prevent deadlock, only four philosophers (or one less than the total number of plates) are allowed into the room at any time to guarantee to be at least one philosopher who is able to make use of both, his left and his right chopstick. In [Carriero and Gelernter, 1990a] this is shown with the *pigeonhole principle*.

The C-Linda solution does not guarantee fairness of choice, it relies on a fair runtime system. With the fairness assumptions made in Sect. 5.4.4 we can prove that no individual starvation can occur, i.e., the solution is fair:

Sketch of proof: If a philosopher wants to eat, he tries to fetch a room ticket:

- If there is a matching one available he will fetch it and continue.
- If there is no one available he will be suspended because all the other philosophers have fetched one. The others will eventually finish eating (general assumption for critical sections) and thus return the room tickets after finite periods of time. The suspended philosopher will *eventually* obtain a room ticket, since the tuple-space manager handles matching between new deposited tuples and templates of suspended processes in a fair way (Sect. 5.4.4).

The same fairness assumptions apply to fetching the chopsticks, thus a philosopher who wants to eat will eventually start eating. This is a sufficient assumption to avoid individual starvation. □

The possibility of a particular philosopher being starved to death by a conspiracy of his two neighbors, which is contained in the solution in [Dijkstra, 1971], is not present in our solution. Note that eating is considered to be a critical section which will eventually terminate and that a fair process scheduler is presupposed. We sketch some constraints that solutions for selective mutual exclusion problems must satisfy:

- Deadlock freedom.
- Fairness.

```
program DiningPhilosophers;
   visible constant n := 5,                     -- #philosophers
                    TS := CreateTS(om);
begin
   for i in [ 0 .. n-1 ] do
      deposit [ "chopstick", i ] at TS end deposit;
      if i /= n-1 then
         deposit [ "room ticket" ] at TS end deposit;
         || closure phil(i); -- spawn a philosopher
      end if;
   end for;
   phil(n-1); -- the main program becomes the last philosopher

   procedure phil (i);
   begin
      loop
         think ();
         fetch ( "room ticket" ) at TS end fetch;
         fetch ( "chopstick", i ) =>
                 fetch ( "chopstick", (i+1) mod n ) at TS end fetch;
           xor ( "chopstick", (i+1) mod n ) =>
                 fetch ( "chopstick", i ) at TS end fetch;
            at TS
         end fetch;
         eat ();
         deposit [ "chopstick", i ] at TS end deposit;
         deposit [ "chopstick", (i+1) mod n ] at TS end deposit;
         deposit [ "room ticket" ] at TS end deposit;
      end loop;
   end phil;
end DiningPhilosophers;
```

Figure 6.8: Solution for the dining philosophers problem.

- Processes that fail within their non-critical sections do not prevent others from eating. A process may halt in its non-critical section. It may not halt during execution of its protocols or critical section. If one process halts in its non-critical section, it must not interfere with other processes [Ben-Ari, 1990; Andrews, 1991].
- Symmetry, in that all philosophers obey precisely the same rules. However, the system has sometimes to prefer specific philosophers to guarantee fairness. Perfectly symmetrical solutions to problems in concurrent programming are impossible because if every process executes exactly the same program, they can never 'break ties' [Ben-Ari, 1990].
- Concurrency (non-neighboring philosophers may eat simultaneously).
- No assumptions concerning the relative speeds.
- Efficient behavior under absence of contention.

6.6 Summary

We extended the informal presentation of generative communication in ProSet of Chap. 5 by means of some example programs. We will continue with a discussion of other proposed approaches to extending Linda and some design alternatives in Chap. 7.

Chapter 7

Discussion of Other Approaches to Generative Communication and Some Design Alternatives

In this chapter we discuss other approaches to extending Linda proposed in the literature and indicate some design alternatives for generative communication in ProSet as proposed in Chap. 5. In the next two sections we begin with a sketch of the shortcomings of C-Linda and compare C-Linda's tuple-space operations with ProSet's tuple-space operations. We then proceed with discussions on:

- Customized matching (Sect. 7.3)
- Selective matching (Sect. 7.4)
- Aggregate and accumulative matching (Sect. 7.5)
- Limited tuple spaces and non-blocking matching (Sect. 7.6)
- Fair matching (Sect. 7.7)
- Extending the type system for matching (Sect. 7.8)
- Formals and templates with first-class rights (Sect. 7.9)
- Update operations on tuples in tuple space (Sect. 7.10)
- Process creation and process identities (Sect. 7.11)
- Multiple tuple spaces and tuple-space identities (Sect. 7.12)
- Data parallelism in generative communication (Sect. 7.13)
- Fault-tolerance (Sect. 7.14)
- Persistence (Sect. 7.15)
- Overloading predefined operators (Sect. 7.16)

7.1 Shortcomings of C-Linda

Despite its strength, as discussed in Sect. 3.2, the C-Linda model of generative communication is not without its weaknesses. Some of the concerns are (see also [Kaashoek *et al.*, 1989; Bal, 1992; Wilson, 1993]):

- Difficult data structures such as sets are not easy to store in tuple space.

  This is no problem in ProSet.

- The expressiveness is limited: some operations which are simply expressed in other systems — in particular update operations — are difficult to express using C-Linda's tuple-space operations.

  ProSet's `meet` operation provides arbitrary update operations on tuples in tuple space to solve this problem.

- C-Linda's global tuple space does not support modularity. This causes disadvantages for information hiding and optimizations.

  Multiple tuple spaces solve this problem.

- An associative tuple space cannot deliver as high a performance as, for example, direct message passing.

  If an application actually needs the functionality of associative addressing, then the cost of it must be paid, no matter what kind of system is being used. The advantage of generative communication is that programmers do not have to implement the mechanisms of associative addressing themselves.

- The performance is unpredictable (see also Sect. 11.4).

- The behavior is not well-defined: there is not a standard semantics for the tuple-space operations in C-Linda.

  We solve this problem by providing a formal specification of generative communication in ProSet in Chap. 8.

7.2 Comparison of ProSet's and C-Linda's Tuple-Space Operations

After sketching some shortcomings of the tuple-space operations in C-Linda and indicating partial remedies, we now compare the tuple-space operations in ProSet (see Sect. 5.4) with the tuple-space operations in C-Linda.

The `deposit` operation comprises the `out` and `eval` operations of C-Linda. One might compare depositing of active tuples by `deposit` with C-Linda's `eval`, but this is not the same, however, because all fields of an `eval` tuple are executed in parallel and not only fields which were selected by the programmer. This is a noteworthy difference: according to the semantics of `eval` *each* field of a tuple is evaluated in parallel. But probably no system will create a new process to compute, for example, a plain integer constant. In the Yale Linda Implementation, only expressions consisting of a single function call are evaluated within new processes [Carriero and Gelernter, 1990a]. The system has to decide which fields to compute in parallel and which sequentially. Similar problems arise in automatic parallelization of functional languages: here one has to reduce the existing parallelism to a reasonable granularity. In our approach the programmer communicates his knowledge about the granularity of his application to the system.

Additionally, the semantics of `eval` is not well-defined and, therefore, not always understood: some current implementations in fact evaluate all fields of an `eval` tuple sequentially within a single new process. This may cause deadlocks if processes within an `eval` tuple communicate with each other. [Narem, 1989] summarizes four basic types of process creation used in implementations of C-Linda's `eval` operation. They are different interpretations of the informal specification of the `eval` operation. Furthermore, it is not well-defined what has to happen if processes, which were spawned by `eval`, perform side effects on global variables. See also [Leichter, 1989, Sects. 3.3 and 3.4.1] for a discussion on this subject.

The `eval` operation — i.e., process creation — was not included in early C-Linda versions [Gelernter, 1985]. Process creation and generative communication are conceptually different. The `eval` operation has been added later to encompass process creation within the tuple-space operations. We did not follow this approach in ProSet to retain orthogonality between process creation and generative communication. In a similar approach it has been proposed to replace `eval` in PROLOG-Linda by parallel committed choice logic [Anderson *et al.*, 1990] (see also Sect. 3.2.4).

To conclude this section, we note that the `fetch` operation combines the `select` construct of Ada [Barnes, 1984]. and `in` respectively `inp` of C-Linda. The `meet` operation combines `select` of Ada, `rd` and `rdp` of C-Linda, and allows for changing tuples in tuple space. Note, however, that a template may be empty in ProSet to match the empty tuple `[]`. This is neither allowed for tuples nor for templates in C-Linda.

7.3 Customized Matching

The Boolean expression after `|` within templates may be used in ProSet to customize matching by restricting the set of possibly matching tuples. Several other approaches to customize the matching process were proposed:

- [Schoinas, 1991] and [Anderson and Shasha, 1992] provide *range accessing*: formals are extended to match only values in a specified range (for example: `in(? i [10:20]);`).
- [Abarbanel, 1991] provides so-called *satisfies forms*, viz. *predicate functions*, which are attached to formals to be evaluated during the matching process.
- Boolean expressions are attached in [Anderson *et al.*, 1990] to the Linda operations in PROLOG-style to customize matching.
- In [Jagannathan, 1991] individual tuple spaces are accessed indirectly via so-called *policy closures*, which may customize the matching process.
- [Broadbery and Playford, 1991] allow to redefine matching methods.

Range accessing may be implemented in ProSet through appropriate conditions, for example:

```
fetch (? i | (10 <= $(1) and $(1) <= 20) ) end fetch;
```

The approaches in [Abarbanel, 1991; Anderson *et al.*, 1990] are similar to our approach. The approaches in [Jagannathan, 1991; Broadbery and Playford, 1991] go beyond our approach. However, the programmer needs knowledge about the internal representation of policy closures and matching methods, respectively.

7.4 Selective Matching

The `fetch` and `meet` operations allow to specify multiple templates in one statement. This way a process can easily wait for one of multiple events. We call such a behavior *selective matching*. Selective matching supports, for example, distributed implementations of backtracking, such as branch-and-bound applications, where selective waiting for multiple events is often desired [Kaashoek *et al.*, 1989].

Other proposals for selective matching operations were made:

- [Abarbanel, 1991] proposes so-called *tuple grouping* which provides a selective matching facility very similar to out approach.
- The Linda Program Builder [Ahmed and Gelernter, 1992] supports a so-called `or-in` *abstraction* for plain C-Linda. This is no language extension, but rather a simulation of selective matching through multiple `in`s and C's `switch` statement [Kernighan and Ritchie, 1988]. The editor supports a program template in which the `or-in` becomes an `in` of a bit vector to check which tuples may be available. This `in` is followed by a conditional which checks which bit is on, and based on that, reads the appropriate tuple. The bit vector has to be generated whenever one of the tuples of the `or-in` is used in an `out` or `eval`. The `or-in` abstraction intends to enable programmers to use and test a proposed language extension before it has actually been added to C-Linda.

7.5 Aggregate and Accumulative Matching

We considered adding the facility for specifying multiple templates in a list to avoid unnecessary sequences for fetching and meeting tuples as in

```
meet ( "name1", ? x1 ) and ( "name2", ? x2 )
               => put("name1 and name2 met");
  at TS
end meet;
```

But it is intuitively not obvious, whether both templates have to match tuples at the same time or during an interval of time (between starting and finishing the `meet` or `fetch` operation). For instance, in [Anderson and Shasha, 1992] multiple templates are allowed for `rd` and `rdp` operations, which set read-locks on the involved tuples to guarantee simultaneous matching. Conversely, in [Broadbery and Playford, 1991] accumulate matching for multiple templates is discussed, where matching must take place between starting and finishing the respective operation: the involved tuples do not have to be at the same time in the tuple space. Due to the unclear semantics we decided not to support aggregate and accumulative matching; the above statement is not allowed in ProSet.

7.6 Limited Tuple Spaces and Non-blocking Matching

There exists some (minor) vagueness concerning the notions of limited tuple spaces and non-blocking matching. Firstly, the meaning of raising `ts_is_full` when a tuple space is full as introduced in Sect. 5.4.1 is not really satisfying: while the handler is executed, there might already be some space available. Blocking on full tuple spaces is new to pure tuple-space communication. However, we prefer to make the programmer aware of the problem of finite memory, instead of hiding it in the system.

Most runtime systems for Linda hide the fact of limited memory from the programmer. For instance, in [Hutchinson, 1990] `out` is suspended until space is available again. This might cause deadlocks which otherwise would not occur. Conversely, programs running with the runtime system from Scientific Computing Associates print the message "`out of tb's`" and exit when the tuple space is exhausted [Sci, 1992].

[Carriero and Gelernter, 1990a, page 112] propose a "high watermark/low watermark" approach to control the amount of sequence data in tuple space with C-Linda. This approach requires additional explicit synchronization with extra tuples. We argue that ProSet's limited tuple spaces provide a higher-level approach to balancing the amount of data in tuple space, because the programmer simply specifies a limit on the expected or desired size of a tuple space instead of providing additional synchronization.

Secondly, there exist also problems with non-blocking matching. The non-blocking operations `inp` and `rdp` of C-Linda are considered harmful by some people (see e.g. [Leichter, 1989, Sect. 3.5]). Especially in a distributed environment matching tuples may become available while the `else` statements of a non-blocking `fetch` or `meet` operation are executed in ProSet. This is so because of concurrent access to tuple space. Thus the meaning of these statements is not really satisfying. Generally it is hardly possible to make any meaningful statement on the *state* of a tuple space, like "This tuple is not in tuple space!". The state can change while it is checked. It is recommended not to use non-blocking matching, if the vagueness implied constitutes a problem.

As one can see in the examples of Sects. 6.1 and 6.2, we consider non-blocking matching useful to handle limited tuple spaces, because we do not know the exact number of results we had to fetch. We argue that the semantic problems with supporting the notion of limited tuple spaces and non-blocking matching are related. Our conclusion:

> We should support the notion of limited tuple spaces if and only if we also provide non-blocking matching.

The specification of a limit for a tuple space is primary intended here to obtain a facility for load balancing between producer and consumer. If we regard limited tuple spaces as a device to influence load balancing, then the vagueness introduced by limited tuple spaces and non-blocking matching is not a problem: one deposits *some* tasks, thereupon one fetches *some* results, and so on. This leads us to the following conclusions:

- Limited tuple spaces and non-blocking matching appear to be useful for handling load balancing between processes.
- In applications where these vagueness might be a problem, one should not use non-blocking matching, not specify limits for tuple spaces, and not customize the behavior on reaching full tuple spaces.

7.7 Fair Matching

Weakly fair selection of pending processes has been specified for ProSet in Sect. 5.4.4. Deposited tuples are selected without any consideration of fairness. Since deposited tuples are no longer associated to processes, it is reasonable to select them without any consideration of fairness. Linda's semantics does not guarantee tuple ordering — this aspect remains the responsibility of the programmer. If a specific order in selection is necessary, it has to be enforced via appropriate tuple contents.

If we would guarantee strongly fair selection of templates then the system would have to retain non-blocking matching operations of processes, for which no matching tuples were available. However, a requesting process cannot be put into a queue because it does not wait: we cannot enable an executing process. One has to be aware that busy waiting with polling methods, which use non-blocking matching operations for example in loops, are *not* handled in a fair way.

Other proposals for fair matching operations were made:

- [Leichter, 1989, section 10.4] proposes probabilistic tuple-space operations with random selection of tuples (not templates). This is probabilistic fairness according to [Kwiatkowska, 1989].

  This approach is somewhat the opposite to our approach, since it specifies fair selection of tuples and not fair selection of templates.

- [Sutcliffe, 1993] uses a wait queue for the implementation of `in` and `rd` requests which could not be satisfied immediately.

  This approach is very similar to the approach in our implementation design in Chap. 9.

[Dijkstra, 1988] argues that "fairness, being an unworkable notion, can be ignored with impunity" because finite experiments cannot distinguish between fair and unfair implementations. Conversely, we view fairness as a simplifying assumption to increase abstraction. Concepts that cannot be verified by finite experiments, such as liveness properties, are introduced to make program design simpler. Fairness is such a liveness property, which allows us — among other things — to reason about program termination in the presence of nondeterminism. Another approach to abstraction is the use of real numbers to specify numerical calculations: even though they cannot be represented exactly by computers, real numbers provide a convenient language for describing calculations which the computer will carry out approximately.

To emphasize the practical relevance of fairness in generative communication, we quote Bob Bjornson's experience with an unfair implementation of C-Linda:

> "When this program was first run, we noticed a very strange behavior: some workers received no tasks, while others received theirs with no apparent contention. As the granularity was decreased, the number of 'starving' workers increased. This marked unfairness was due to the way templates were stored in tuple space: last-in-first-out queues. As contention arose, some templates languished at the bottom of the queue, and the workers that had issued them starved. Partly motivated by this experience, we chose to store templates first-in-first-out, making `in`s and `rd`s fairer." [Bjornson, 1992, page 57]

Note, however, that this is an informal fairness notion.

7.8 Extending the Type System for Matching

As Linda relies heavily on type matching, the type system of the computation language has a notable effect on tuple-space semantics and implementation. For example in C, the equivalence of types is not that obvious. Under which conditions are structures respectively unions equivalent? Are pointers equivalent to array-names? See for example [Narem, 1989] for an informal discussion of type equivalence in C-Linda. In [Leichter, 1989], it has been proposed to extend the type system of C to overcome some of the problems thus caused: each expression has two distinct types associated with it, its *C type* and its *Linda type*. The Linda type follows stricter rules and is significant only in tuple matching, thus these type extensions only influence the matching process and not the type system of C.

In ProSet there is no necessity for extending the type system for obtaining a smooth integration of Linda: firstly, since ProSet provides a well-formed type system with clear semantics for type equivalence, there exists no necessity to extend the basic type system for tuple matching. Secondly, since there exist no difference between ProSet-tuples and passive Linda-tuples, a combination on the basis of this common feature becomes straightforward.

7.9 Formals and Templates with First-Class Rights

One could introduce a new data type for formals. Tuples that contain formals would be templates, thus making formals and templates first-class objects. However, objects of these types would only be useful in tuple-space operations, and therefore it does not seem to be justified to introduce such new types. Formals in deposited tuples would extend the matching rules: such formals would match with appropriate actuals in templates, but never with formals in templates. For the time being, we see no substantial advantages of such an extended matching procedure.

Conversely, in C-Linda formal fields may be `out`ed. When used, matching will be performed using the usual rules, but no assignment is made for that field [Narem, 1989]. This is inconsistent, since formals in tuples are not treated in the same way as formals in templates.

7.10 Update Operations on Tuples in Tuple Space

C-Linda does not provide update operations on tuples in tuple space. To simulate updates, a tuple must be removed and the updated tuple must be inserted. To increment a shared counter, for instance, the following sequence of operations has to be programmed:

```
in ( "counter", ?  i );
out ( "counter", i+1 );
```

This approach has two disadvantages:

1. The programmer cannot directly display his intention: he wants to update a shared datum, but has to remove an old datum and insert a new one.
2. In-place updates would be implemented more efficiently (see also Sect. 11.3.2).

What the programmer usually intends is an atomic modification of the tuple contents; removing and replacing the tuple, though inefficient, is the only way to accomplish this in C-Linda.

ProSet provides arbitrary update operations on tuples in tuple space to overcome these shortcomings (see Sect. 5.4.3). Some other proposals for adding in-place update operations to Linda were made:

- In a similar approach, [Anderson and Shasha, 1992] propose an `in-out` operation for accomplishing in-place updates.
- [Wilson, 1993] proposes in-place update operations on tuples in tuple space, but restricts them to scalar operations. This restriction does not remove the absence of support for user-defined high-level operations on shared data in Linda [Bal, 1992], which is removed in ProSet.

7.11 Process Creation and Process Identities

In C-Linda, there exists an inherent distinction between at least two classes of processes. Processes live inside and outside of tuple space: the main program is not part of an active tuple (thus it lives outside of tuple space), and all additional processes are created via `eval` as part of active tuples hence they live inside the tuple space.

But often it is not desired to put the return values of spawned processes (if after all available) into tuples in tuple space. This is for instance the case if a worker process executes in an infinite loop and deposits result tuples into a tuple space instead of returning only one result. It seems to be artificial to put such a worker process into an active tuple. In ProSet, the approach for process creation known from Multilisp is adapted to set-oriented programming (Sect. 5.1). Here, new processes may be spawned inside and outside of tuple space. This way, process creation and tuple-space communication became *orthogonal* concepts in ProSet.

Processes have *no* first-class rights in ProSet. As ProSet uses value semantics, it must be possible to obtain copies of objects with first-class rights, but running processes can exist only once. It is not possible to get a copy of a running process: the copied process may do other things than the original process because of race conditions. Both processes (original and copy) would not be identical any longer contradicting the requirement that the value of a copy has to be equal to the original value after copying.

One could introduce process identities with first-class rights that would be returned by the process creator `||`. The programmer could test, if the value is evaluated or not. After evaluation, the returned value would replace the process identity. It would be necessary to replace *all* copies of such a process identity. This approach is proposed in [Leichter, 1989]. We quote one important characteristic of futures as an agreeable argument against such an approach:

> "Also, no special care is required to use a value generated by `future`. Synchronization between the producer and the user of a future's value is implicit, freeing the programmer's mind from a possible source of concern." [Halstead, 1985, page 505]

We decided not to sacrifice this simplicity. Other proposals for combining Linda with futures were made:

- BaLinda-Lisp [Yuen *et al.*, 1993] provides a notion for *futures* for process creation, but this is like a fork in Unix. A *touch* does not guarantee a deterministic result in BaLinda-Lisp, because blocking is not applied when a future is touched.
- [Wilson, 1993] proposes *future*-like variants of C-Linda's `in` and `rd` operations, but restricts this enhancement to these operations.
- [Landry and Arthur, 1992] propose to automatically replace `in` and `rd` operations based on compile-time analysis by their respective *future*-like variants. These *future*-like variants of the `in` and `rd` operations are similar to the ones proposed in [Wilson, 1993], but not explicitly available to the programmer.

7.12 Multiple Tuple Spaces and Tuple-Space Identities

The idea of splitting the tuple space into multiple spaces is frequently applied. New data types or classes are often proposed to organize them [Matsuoka and Kawai, 1988; Leler, 1990; Abarbanel, 1991; Ciancarini, 1991; Wilson, 1991b; Zettler, 1992, and many others]. Some advantages of multiple tuple spaces over the flat tuple space in C-Linda are:

- Multiple tuple spaces allow the programmer to partition the communication medium according to the application.
- The representation of individual tuple spaces may be customized based on their contents.
- The compile-time analysis is simplified with respect to partitioning the tuple space (see also Sect. 11.3.1).
- Individual tuple spaces may obtain different properties. For example, a replication, which makes read operations cheap and write operations more expensive, could be chosen for selected tuple spaces. Read operations could be done locally on a processor without the need for communication (see also Sect. 11.4).
- Information hiding is supported.

The small examples presented in Chap. 6 did not fully demonstrate the advantages of manipulating multiple tuple spaces. However, in more sophisticated problem domains such as process trellises [Factor, 1990] the advantage of information hiding is obvious, since processes may communicate within isolated tuple spaces independent of communication in other tuple spaces.

In most proposals for introducing multiple tuple spaces, values of objects of these types are usually used as identifiers/references to tuple spaces and not as the value of a tuple space itself. These approaches may be compared with our approach to use atoms to identify tuple spaces: in [Jagannathan, 1991] the function `make-ts` returns a reference to a *representation structure* of a newly created tuple space. Group identifiers are used in [Anderson, 1991] as optional parameters for the tuple-space operations to split the tuple space into multiple groups. If no group identifier is specified, the operation applies to the *default* tuple space, which is a global, flat tuple space as in plain C-Linda.

In [Gelernter, 1989] and [Hupfer, 1990] multiple, first-class tuple spaces are hierarchically structured like files in Unix. Current default tuple spaces are provided similar to the *current working directory* in Unix. In addition to using *path names* of tuple spaces, operations on tuple spaces as first-class objects are supported (for example suspension). Concerning the data type `ts` in Melinda, Susanne Hupfer writes:

> "Though the type `ts` is an essential theoretical component in Melinda, and we can have `ts`-typed variables, we have no physical conception of a `ts`-typed object, no operations that we can perform upon it, and indeed no way of representing the value of such an object textually, for example as a `ts` constant ..." [Hupfer, 1990, page 12]

Conversely, [Ciancarini, 1991] provides tuple-space constants for the creation of tuple spaces.

However, because of concurrent access it is rarely possible to make any sensible statement with respect to the *actual value* of a tuple space. A tuple space may be viewed as the dynamic envelope of a growing and shrinking multiset of passive and active tuples that controls the communication and synchronization of parallel processes. This dynamic communication device has *no* first-class rights in ProSet. Atoms as tuple-space identities already have first-class rights. We see no necessity and justification to introduce a *new* data type with first-class rights for tuple-space identities in ProSet.

There remains another important question concerning multiple tuple spaces: How to organize multiple tuple spaces? Hierarchical tuple spaces are proposed in [Gelernter, 1989; Hupfer, 1990; Wilson, 1993]. This organization is superior to flatly organized multiple tuple spaces if strings are used to *open* the tuple spaces for access. If one allows unrestricted navigation in hierarchical tuple spaces, which are opened via string handles, one will not have real information hiding (but structured, global information). If the same strings are used accidentally by two software components to open tuple spaces on the same level of a hierarchy, this may cause problems. Conversely, if one has something like ProSet's atoms to identify tuple spaces, it is not necessary to restrict the organization of multiple tuple spaces to hierarchies. The programmer can organize multiple tuple spaces as he sees fit, and real information hiding is possible.

A tuple-space identity may be regarded as a *capability* [Tanenbaum, 1992, Sect. 4.5.3] to a shared associative memory. Processes that have access to this capability can perform operations on this shared memory. This model suggests a non-hierarchical relation among tuple spaces: the elements of a tuple space are tuples whose component fields are never tuple spaces themselves. A tuple can contain the identity of a tuple space, never the tuple space itself.

The absence of any rigid structural constraints on the contents and organization of the tuple spaces has other implications as well. Since tuple spaces contain processes as well as passive data, there is no process structure imposed on a ProSet program: such programs can configure themselves automatically and dynamically based on the number of tuple spaces created and the contents found within them. To build a set of related processes, the processes are encapsulated within the same tuple space. Data values that need to be accessed by a known collection of processes are deposited within a tuple space accessible only to these processes.

The motivation for incorporating multiple tuple spaces is in part technical and in part conceptual. By permitting tuple spaces to be named, we allow the programmer to partition the communication medium as is appropriate. Partitioning of this kind permits the compiler in turn to customize the representation of individual tuple spaces based on their contents. Of course, a clever optimizing compiler can infer a partitioning scheme of a designated field within a tuple to serve the role of a hash key — all tuples containing the same hash key would be placed in the same partition of tuple space (see also Sect. 11.3.1). Multiple tuple spaces simplify the complexity of compile-time analysis in this respect. Moreover, the specification of the optimizations underlying the customization of a tuple space becomes amenable to formal description, because tuple spaces themselves constitute a distinguished type: the same analysis underlying the optimization and use of other data types in the base language can be applied to tuple spaces as well.

Other proposals for enhancing generative communication with information hiding have been made:

- [Pinakis, 1992] introduces information hiding through *directed communication* in tuple space similar to Internet connections via pairs of Internet address and port number. New types for tickets and addresses are introduced, which must not be used with formals while matching. This approach is similar to the Amoeba distributed operating system's implementation of public and private ports [Tanenbaum *et al.*, 1990].
- The approach in [Tolksdorf, 1992] is similar to the approach in [Pinakis, 1992], but the tickets are not visible to the processes. They are inserted by the runtime system.
- In [Patterson *et al.*, 1992], a `hide` interface routine is proposed, which gives an application the capability to make a tuple or group of tuples *invisible* to other processes.

7.13 Data Parallelism in Generative Communication

Data parallelism is discussed as an approach to parallel programming in Sect. 3.2.1. It has been proposed to combine generative communication with data parallelism. As one instance, [Anderson and Shasha, 1992] propose operations on *all* tuples in tuple space which match a given template:

```
int i, sum = 0;
...
inp ("count", ? i) all { sum += i; };
```

This example would sum the second field of every tuple in tuple space matching the template, including duplicates. The code in the block of the command is executed, with the formal-variables bound to the matching tuple's values, each time a new tuple is retrieved. Except for the guarantee that each tuple is retrieved only once, the semantics is that of a loop with the tuple-space operator at the head and the code block as the body:

```
int i, sum = 0;
...
while (inp ("count", ? i)) { sum += i; };
```

Conversely, in most data-parallel systems, data-parallel operations appear to be done *simultaneously* on all affected data elements (see Sect. 3.2.1). Consequently, we anticipate serious problems with adding data-parallel operators to generative communication systems concerning the semantics:

> What happens if multiple processes try to change the same set of tuples simultaneously? Is the entire set locked by one process for the duration of the data-parallel operation or are the tuples locked successively?

One could define it in both ways, but for us it is not clear which one is the *natural* way.

[Abarbanel, 1991] proposes variations of `in` and `rd` that return *all* tuples in tuple space which match, but in an atomic transaction (as opposed to [Anderson and Shasha, 1992], where the semantics is that of a loop). Rob Abarbanel mentions the following problem thus caused:

"These functions conflict with the simple Linda model. They introduce a concept of time. They return all the tuples that match at the time of the transaction. The basic Linda model has no such concept. In fact, the ability to program without explicit handling of orderings of events is one of the best features of Linda. In basic Linda, there is no way to tell whether a returned tuple was produced in the past, or even in the future." [Abarbanel, 1991, page 13]

[Leichter, 1989, Sect. 10.1.3] sketches a complicated implementation technique for *loops over all tuples matching some template*. In this proposal, the iteration proceeds successively and not atomically. [Anderson and Shasha, 1992] make the following remark concerning such an approach:

"The **all** variation of a command is *not* atomic; the set of tuples that match the pattern can change before the command completes. We could have used 'snapshot' semantics requiring that when the command is issued all tuples currently matching the pattern are retrieved, but we felt that this was too difficult to implement efficiently. We are interested in reactions of potential users." [Anderson and Shasha, 1992, pages 104–105]

Because of these problems we do not consider to extend generative communication in ProSet with data-parallel operations. Such an extension would cause serious problems both for the definition of the semantics and for the implementation.

7.14 Fault-tolerance

Fault-tolerance for implementations of Linda has been considered:

- In [Xu and Liskov, 1989], fault-tolerance is achieved by a replication of tuple space on a small subset of available processors in a network. The replication is completely hidden from the programmer. Tuple space is uniformly replicated, that is, each replica contains an entire copy of the tuple space. Fault-tolerance is guaranteed with respect to node crashes and network failures.
- In [Kambhatla and Walpole, 1990], data is made highly available through replication, and processes are made recoverable through periodic checkpointing of process states.
- In [Patterson *et al.*, 1992], tuples are replicated among so-called subspaces of the tuple space. The tuple space is not uniformly replicated.

These approaches to fault-tolerance in Linda assume fail-stop processors, wherein a processor may suddenly halt and kill all executing processes. This concept excludes so-called Byzantine failures where processes malfunction and emit spurious and even contradictory results.

[Bakken and Schlichting, 1993] propose a *conditional atomic tuple swap operator* to facilitate fault-tolerance in master-worker applications. In the following statement:

`in(`*template*`)` ⇒ `out(`*tuple*`);`

the `out` operation is executed atomically with the `in` operation, that is, in such a way that there is no visible intermediate state of the tuple space that contains the result of the `in` operation, but not the result of the `out` operation. Arguments assigned by the call of the `in` operation are available to the call of the `out` operation. The programmer must explicitly use such atomic swap operations in the worker processes, and it is necessary to write a monitor process which regenerates subtask tuples of failed workers. A stable Linda implementation as proposed in [Xu and Liskov, 1989; Kambhatla and Walpole, 1990; Patterson *et al.*, 1992] is assumed, which can survive detectable processor failures suffered by fail-stop processors that execute the worker processes.

Note that such an atomic `in-out` operation has some ideas in common with changing tuples via ProSet's `meet` operation (Sect. 5.4.3). Note also that for a prototyping language fault-tolerance is a minor concern (see also Sect. 3.1).

7.15 Persistence

The addition of persistence to generative communication has been considered. For instance, in [Anderson and Shasha, 1992] the tuple space is made persistent, i.e., it survives the execution of an application program.

Tuple spaces in ProSet are not persistent. They exist only until all processes of an application have terminated their execution. Consequently, tuple-space communication in ProSet is designed for *multiprocessing* (single application running on multiple processors) as opposed to *multiprogramming* (separate applications).

Multiprogramming in ProSet is programmed with a mechanism for handling persistent data, which is based on critical sections on the persistent data values (see Sect. 4.5). Persistent data values survive the execution of an application program. A typical application domain for multiprogramming is a database system. Database systems contain persistent data.

Generative communication in ProSet is designed to enable multiprocessing, where multiple processes execute as components of a single application program (logical parallelism). These multiple processes may execute on multiple processors (physical parallelism). See Sect. 3.1 for a definition of logical and physical parallelism.

In multiprocessing, multiple processes solve a problem cooperatively, whereas in multiprogramming, multiple programs work to a large extent independently and access only occasionally shared data. The access to shared data in multiprogramming is competing rather than cooperating. Consequently, critical sections on the persistent data values are an appropriate means for managing the access to these values and retaining a consistent persistent store.

For multiprocessing, a flexible device for communication is needed. A tuple space is a multiset of tuples and not a set of tuples. Because of concurrent access by the cooperating processes to tuple space, it is necessary to have multisets and not sets for coordination: it is not easy and even not desired for coordination to guarantee the uniqueness of tuples. Multisets are, therefore, a suitable data structure for communication between cooperating processes, because the data flow is not restricted unnecessarily. Consequently, multisets allow a high degree of parallelism for cooperating processes.

Access to tuple space is associative. This allows a high degree of decoupling between the cooperating processes. Conversely, ProSet's persistent data values are accessed by name. To conclude this section we note that in ProSet there is a smooth separation of concerns for multiprocessing and multiprogramming (both concepts are orthogonal):

- Parallelism is the *exception* for competing programs, therefore coordination is only necessary to retain the consistency of the persistence store in multiprogramming. Critical sections are an appropriate means for avoiding lost updates.
- Parallelism is the *rule* for cooperating processes, therefore coordination needs an efficient and flexible medium in multiprocessing. Tuple spaces are an appropriate device for cooperation.

7.16 Overloading Predefined Operators

It has been considered to replace the tuple-space operations proposed in Sect. 5.4 syntactically through overloading of predefined operators. For instance, a `deposit` operation such as:

```
deposit [...] at TS end deposit;
```

could be replaced by the following assignment:

```
TS := TS with [...];
```

Hence, ProSet's `with` operator would not only be defined for sets and tuples, but also for atoms. ProSet's `with` operator adds an element to a set or tuple [Doberkat *et al.*, 1992a]. As one result of such a replacement, for atoms the following equation would *always* hold:

```
(TS with [...])  = TS
```

This is in general not the case for tuples and sets thus yielding inconsistent behavior for the `with` operator. The reason for this strange behavior is that the `with` operator is applied to a tuple-space identity and not to a tuple space. Note also that, for the time being, the unique binary operation for ProSet's atoms is comparison for equality. However, such a replacement of the `deposit` operation does not produce too many problems, but there are more serious problems with replacing a `fetch` operation such as

```
        fetch (...) at TS end fetch;
```

by, for example,

```
        TS := TS less (...);
```

ProSet's `less` operator removes an element from a set. Two essential problems arise with such a replacement:

1. It would be necessary to have templates with first-class rights. This problem is discussed in Sect. 7.9.
2. To replace `fetch` by `less`, the `less` operator needs the capability to block. For the time being, none of ProSet's predefined operators has the capability to block.

Our conclusions are that overloading existing operators in a computation language for obtaining tuple-space operations may be a nice way to keep the language as small as possible. In ProSet, the existing operators do not seem to be appropriate for overloading them this way. To avoid confusion derived from overloading predefined operators we argue that it is a more appropriate choice introducing the three new tuple-space operations and make them statements syntactically.

7.17 Summary

We discussed other approaches to extending Linda and some design alternatives to our approach which has been proposed in Chap. 5. Therefore, this chapter serves as a motivation for our approach to generative communication and as a discussion of related work concerning the extension of Linda.

Chapter 8

Formal Semantics of ProSet-Linda

This chapter presents a formal specification of generative communication in ProSet by means of the formal specification language Object-Z [Duke *et al.*, 1991]. Appendix B provides a short introduction to the specification language Object-Z. The reader may refer to this appendix. The informal specification of generative communication in ProSet is presented in Chap. 5.

The present presentation is meant to be self-contained and no previous knowledge of Object-Z is required to understand it — at least we hope so. Where necessary, we provide notes to explain the notation used (enclosed in the symbols $\boxed{Z}$ and $\boxed{Z}$). The index of explained Z symbols and keywords at the end of this document refers to these explanations. The presented specification has been developed with the *f*uzz package [Spivey, 1992a]. The *f*uzz package contains a type-checker for plain Z and not for Object-Z. Therefore, the object-oriented extensions to Z have not been type-checked. Appendix C provides a summary of all names defined globally in the specification with their associated types as *f*uzz sees it: the class structure is not visible in this list.

Notational convention: components of ProSet programs are displayed in `typewriter` font to set them apart from Z specifications, which are displayed in *slanted* font. We display identifiers in sans serif font when we use them as infix relation or operation symbols.

8.1 Basic Definitions

The basic, given types are *Expression*, *LValue*, *Process*, *Statement*, and *Value*. Some additional basic types are based on these types and introduced via free type definitions.

8.1.1 Abstractions for the Embedding into the Computation Language

We specify generative communication in ProSet and not the entire language. Therefore, we need for the embedding in the computation part interfaces to some basic concepts of ProSet. At first we need basic types for describing *l*-values and unevaluated expressions:

$$[Expression, LValue]$$

For a detailed discussion of expressions and *l*-values, and their relationship in ProSet we refer to [Doberkat *et al.*, 1992a] (see also Sect. 4.1). Note that *l*-values are not typed. Additional necessary basic types are statements and processes:

$$[Statement, Process]$$

Each process is unique. New processes may be spawned and existing ones may terminate. Note that processes have no first-class rights in ProSet. We shall need a notion for execution of statements:

$$\mathit{Execute}\_ : \mathbb{P}\,\mathit{Statement}$$

$\boxed{Z}$ The underscore _ indicates the position of operands thus *Execute* is an unary predicate. A predicate is identified with the set of objects for which the predicate holds. $\mathbb{P}$ yields the power set of its operand. $\boxed{Z}$

This way we model the execution of statements, and that statements do not yield values in ProSet. There is nothing more to be said about the execution of statements in our formal specification.

8.1.2 Types and Values

We have to know a few specific things about types and values in our specification. The unary operator **`type`** yields a predefined type-atom according to the type of its operand (see Sect. 4.1). The following equations hold in ProSet:

```
    type 1 = integer
  type integer = atom
type type type 1 = atom
```

No particular basic type for the type-names **`atom`**, **`boolean`**, **`integer`**, **`real`**, **`string`**, **`tuple`**, **`set`**, **`function`**, **`modtype`**, and **`instance`** is needed in our Z specification. ProSet does not employ the type matching known from C-Linda and similar embeddings of Linda into statically typed languages. Instead, *conditional value matching* as described in Sects. 5.4.2 and 8.2 is employed. It is sufficient to model types in ProSet through the **`type`** operator and the predefined type-atoms, which are values:

$[\mathit{Value}]$

Values have the following properties in ProSet:

$$\begin{array}{|l}
\mathit{atom}, \mathit{boolean}, \mathit{integer}, \mathit{real}, \mathit{string}, \mathit{tuple}, \mathit{set}, \mathit{function}, \mathit{modtype}, \mathit{instance} : \mathit{Value} \\
\mathit{TRUE}, \mathit{FALSE} : \mathit{Value} \\
\mathit{om} : \mathit{Value} \\
\mathit{ValuesOfType} : \mathit{Value} \rightarrowtail\!\!\!\!\!\!+\; \mathbb{P}\,\mathit{Value} \\
\hline
\mathit{ValuesOfType}\ \mathit{boolean} = \{\mathit{TRUE}, \mathit{FALSE}\} \\
\mathrm{dom}\ \mathit{ValuesOfType} = \\
\quad \{\mathit{atom}, \mathit{boolean}, \mathit{integer}, \mathit{real}, \mathit{string}, \mathit{tuple}, \mathit{set}, \mathit{function}, \mathit{modtype}, \mathit{instance}\} \\
\mathrm{dom}\ \mathit{ValuesOfType} \subset \mathit{ValuesOfType}\ \mathit{atom} \\
(\{\, \mathit{om} \mapsto \{\mathit{om}\}\,\} \cup \{\, t : \mathrm{dom}\ \mathit{ValuesOfType} \bullet t \mapsto \mathit{ValuesOfType}\ t \,\})\ \mathsf{partition}\ \mathit{Value}
\end{array}$$

$\boxed{Z}$ $X \rightarrowtail\!\!\!\!\!\!+\; Y$ is the set of partial injections from X to Y. dom yields the domain of a relation. A set S is a *proper subset* of a set T ($S \subset T$) if every member of S is also a member of T and if in addition S is different from T. The subset relation symbol is $S \subseteq T$. The notation $x \mapsto y$ is a graphical way of expressing the ordered pair (x, y). $\boxed{Z}$

The Boolean values are **`true`** and **`false`** as usual. We use capital letters for *TRUE* and *FALSE* in our specification, because *true* and *false* are predefined in Z. Every value in ProSet, except for **`om`**, belongs to exactly one type set (defined by the last property). Each type atom is mapped by *ValuesOfType* to the set of values which belong to the type it denotes. The type atoms themselves are atoms. The undefined value **`om`**, which indicates undefined situations, has no type. Applying the unary operator **`type`** to **`om`** is undefined, and thus yields the undefined value (`type om = om`). The corresponding function is *Type*:

$$\begin{array}{|l}
Type : Value \longrightarrow Value \\
\hline
boolean = Type\ TRUE = Type\ FALSE \\
atom = \begin{array}[t]{l} Type\ atom = Type\ boolean = Type\ integer = Type\ real = Type\ string = \\ Type\ tuple = Type\ set = Type\ function = Type\ modtype = Type\ instance \end{array} \\
om = Type\ om \\
\forall x : Value \mid x \neq om \bullet \\
\quad x \in ValuesOfType\,(Type\ x)
\end{array}$$

$\boxed{\text{Z}}$ $X \longrightarrow Y$ is the set of total functions from X to Y. $\boxed{\text{Z}}$

For our purposes it is not necessary to specify the types of ProSet through an additionally given, basic type. It is sufficient to specify the semantics of the `type` operator. The remainder of our specification would not change if we remove or add some type names (except for *atom*, *boolean*, *integer*, and *tuple*).

Partial functions modeling the evaluation of expressions and the return values of processes, respectively, shall be needed later:

$$\begin{array}{|l}
Evaluate : Expression \rightarrow\!\!\!\!\!\!+\ Value \\
ProcRetVal : Process \rightarrow\!\!\!\!\!\!+\ Value
\end{array}$$

$\boxed{\text{Z}}$ $X \rightarrow\!\!\!\!\!\!+\ Y$ is the set of partial functions from X to Y. $\boxed{\text{Z}}$

These are partial functions because the evaluation of expressions and processes might not terminate, and thus not produce a result. We also need a relation denoting the assignment of values to l-values:

$$\begin{array}{|l}
\_\ \mathsf{IsAssigned}\ \_ : LValue \longleftrightarrow Value
\end{array}$$

$\boxed{\text{Z}}$ $X \longleftrightarrow Y$ is the set of binary relations between X and Y. $\boxed{\text{Z}}$

For every pair (lhs, rhs), which is related by IsAssigned, an assignment of the value of *rhs* to *lhs* is modeled. We do not need a more detailed specification of assignment in our specification.

8.1.3 Tuples

Tuples in ProSet have their usual mathematical meaning as ordered sequences of values; a value may appear multiply in a tuple, the order of components appearing in the tuple is relevant. Tuple components may be passive values or executing processes in our specification:

$$TupleComp ::= TupleValue\langle\!\langle Value\rangle\!\rangle \mid TupleProcess\langle\!\langle Process\rangle\!\rangle$$

Conceptually a tuple in ProSet is an infinite vector with almost all components equal to the undefined value `om`. Indexing of tuple components starts with the index `1`. The length returned by the `#` operator of ProSet is the largest index of a component different from `om`, thus `#[1,om] = 1` and `#[om,1] = 2` hold (see also Sect. 4.1). Since almost all components in a tuple are equal to the undefined value `om`, tuples have a finite representation and we are able to model active and passive tuples via finite sequences:

$$APTuple == \mathrm{seq}\ TupleComp$$

$\boxed{\text{Z}}$ $\mathrm{seq}\,X$ is the set of finite sequences over X. These are finite functions from $\mathbb{N}$ to X whose domain is a segment $1\,..\,n$ for some natural number $n \in \mathbb{N}_1$. $\mathbb{N}_1 = \mathbb{N} \setminus \{0\}$, where $\setminus$ is the set difference operation. If a and b are integers, $a\,..\,b$ is the set of integers between a and b inclusive. If $a > b$ then $a\,..\,b$ is empty, thus sequences may be empty. $\boxed{\text{Z}}$

Note that only passive tuples are first-class objects in ProSet. However, in our specification for the basic type *APTuple* we do not distinguish between passive and active tuples for simplicity. As we shall see in Sect. 8.2, the matching procedure will distinguish between passive and active tuples.

Note also that the following equations hold in ProSet:

```
  [om] = []
 [1,om] = [1]
[om,1] /= [1]
```

The predefined function $\#$ for finite sets in Z will not work well to specify the unary `#` operator for tuples in ProSet, because the `#` operator yields the largest index of a component different from `om`:

```
#[om] = 0
```

$$\#\langle om \rangle = 1$$

We define the generic function *Arity* instead:

$$\begin{array}{|l}
[X] \\
\hline
Arity : (Value \rightarrowtail X) \longrightarrow (\mathrm{seq}\, X \longrightarrow \mathbb{N}) \\
\hline
\forall\, tup : \mathrm{seq}\, X;\ Xvalue : Value \rightarrowtail X \bullet \\
\qquad Arity\, Xvalue\, tup = max(\{0\} \cup \{\, i : \mathbb{N} \mid tup\ i \neq Xvalue\ om \,\}) \\
\hline
\end{array}$$

$\boxed{Z}$ $\{\, i : \mathbb{N} \mid pred \,\}$ is an abbreviation for $\{\, i : \mathbb{N} \mid pred \bullet i \,\}$. max yields the maximum of a set of integers. Note that function application associates to the left in Z, so $f\ x\ y$ means $(f\ x)\ y$. $\boxed{Z}$

Arity is generic, because it applies to tuples in the same way as it shall apply to templates. *TupArity* then works for tuples:

$$\begin{array}{|l}
TupArity : APTuple \longrightarrow \mathbb{N} \\
\hline
TupArity = Arity\ TupleValue
\end{array}$$

TupArity yields the largest index of an *APTuple* component which is different from the *TupleValue* of *om*.

8.1.4 Formals and Templates

To model formals we need auxiliary type definitions for optional *l*-values and optional `into` expressions:

$$OptLValue ::= NoLValue \mid IsLValue\langle\langle LValue \rangle\rangle$$
$$OptInto ::= NoInto \mid IsInto\langle\langle Expression \rangle\rangle$$

Formals then consist of optional *l*-values and optional `into` expressions:

$$\begin{array}{|l}
\underline{\ Formal\ } \\
Destination : OptLValue \\
Into : OptInto \\
\hline
\end{array}$$

Components of templates are formals or values:

$$TempComp ::= TempValue\langle\langle Value \rangle\rangle \mid TempFormal\langle\langle Formal \rangle\rangle$$

Templates consist of a sequence of template components and a conditional expression:

$$\begin{array}{|l}
\hline \textit{Template} \\
\textit{List} : \mathrm{seq}\ \textit{TempComp} \\
\textit{Condition} : \textit{Expression} \\
\hline
\end{array}$$

If no condition is specified in a template, the ProSet-expression `true` is assumed as a condition. We will need the function *TempArity* for matching:

$$\begin{array}{|l}
\textit{TempArity} : \mathrm{seq}\ \textit{TempComp} \longrightarrow \mathbb{N} \\
\hline
\textit{TempArity} = \textit{Arity}\ \textit{TempValue}
\end{array}$$

It applies to template lists in the same way as *TupArity* applies to tuples.

A notion for assignment of tuple components to formals of templates shall be needed:

$$\begin{array}{|l}
\_\ \mathsf{FormalAssign}\ \_ : \textit{Template} \leftrightarrow \textit{APTuple} \\
\hline
\mathbf{let}\ \textit{FormalOf} == \textit{TempFormal}^{-1}; \\
\qquad \textit{ValueOf} == \textit{TupleValue}^{-1}; \\
\qquad \textit{LValueOf} == \textit{IsLValue}^{-1} \bullet \\
\forall\, \textit{temp} : \textit{Template};\ \textit{tup} : \textit{APTuple} \mid \#\textit{temp}.\textit{List} = \#\textit{tup} \land \mathrm{ran}\ \textit{tup} \subseteq \mathrm{ran}\ \textit{TupleValue} \bullet \\
\qquad \textit{temp}\ \mathsf{FormalAssign}\ \textit{tup} \Leftrightarrow \\
\qquad\qquad (\forall\, i : \mathrm{dom}\, \textit{temp}.\textit{List} \mid \textit{temp}.\textit{List}(i) \in \mathrm{ran}\ \textit{TempFormal} \land \\
\qquad\qquad\qquad\qquad\qquad (\textit{FormalOf}(\textit{temp}.\textit{List}\ i)).\textit{Destination} \in \mathrm{ran}\ \textit{IsLValue} \bullet \\
\qquad\qquad \textit{LValueOf}((\textit{FormalOf}(\textit{temp}.\textit{List}\ i)).\textit{Destination})\ \mathsf{IsAssigned}\ \textit{ValueOf}(\textit{tup}\ i))
\end{array}$$

> Z **let** introduces local definitions with nested scoping for predicates or for expressions. R^{-1} is the relational inverse of the relation R. $S.C$ is the notation for selecting a component C from a binding of a schema S. Z

For every pair $(\textit{temp}, \textit{tup})$ which is related by FormalAssign an optional assignment to the *Destinations* of *temp* is modeled. The assignment takes place for each formal which contains an *l*-value. These *l*-values are then assigned the corresponding tuple fields. A notion for evaluation of `into` expressions shall be needed:

$$\begin{array}{|l}
\_\ \mathsf{EvalIntos}\ \_ : (\textit{Template} \times \textit{APTuple}) \longrightarrow \textit{APTuple} \\
\hline
\mathbf{let}\ \textit{FormalOf} == \textit{TempFormal}^{-1}; \\
\qquad \textit{ExprOf} == \textit{IsInto}^{-1} \bullet \\
\forall\, \textit{temp} : \textit{Template};\ \textit{tup}, \textit{newtup} : \textit{APTuple} \mid \#\textit{temp}.\textit{List} = \#\textit{tup} \land \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{ran}\ \textit{tup} \subseteq \mathrm{ran}\ \textit{TupleValue} \bullet \\
\qquad (\textit{temp}\ \mathsf{EvalIntos}\ \textit{tup} = \textit{newtup}) \Leftrightarrow \\
\qquad\qquad \#\textit{tup} = \#\textit{newtup} \land \\
\qquad\qquad (\forall\, i : \mathrm{dom}\, \textit{temp}.\textit{List} \mid \textit{temp}.\textit{List}(i) \in \mathrm{ran}\ \textit{TempValue} \bullet \\
\qquad\qquad\qquad \textit{newtup}\ i = \textit{tup}\ i) \land \\
\qquad\qquad (\forall\, i : \mathrm{dom}\, \textit{temp}.\textit{List} \mid \textit{temp}.\textit{List}(i) \in \mathrm{ran}\ \textit{TempFormal} \bullet \\
\qquad\qquad\qquad \textit{newtup}\ i = \mathbf{if}\ (\textit{FormalOf}(\textit{temp}.\textit{List}\ i)).\textit{Into} = \textit{NoInto} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ \textit{tup}\ i \\
\qquad\qquad\qquad\qquad\qquad \mathbf{else} \\
\qquad\qquad\qquad\qquad \textit{TupleValue}(\textit{Evaluate}\ (\textit{ExprOf}((\textit{FormalOf}(\textit{temp}.\textit{List}\ i)).\textit{Into}))))
\end{array}$$

> Z $X \times Y$ is the Cartesian product of X and Y (a set of pairs). Z

EvalIntos yields from a pair $(\textit{temp}, \textit{tup})$ a new *APTuple* that is equal to *tup* except for the fields in which *temp* has `into` expressions. Those fields are replaced by the corresponding evaluated expression values. Tuple fields for which the corresponding template fields are actuals or formals without `into` expressions remain unchanged.

8.2 Matching

ProSet employs *conditional value matching* as informally specified in Sect. 5.4.2. A tuple and a template match iff all the following conditions hold:

- The tuple is passive.
- The arities are equal.
- Values of actuals in templates are equal to the corresponding tuple fields.
- The Boolean expression after | in the template evaluates to `true`. If no such expression is specified, then `true` is the default.

As a first step we define matching of individual tuple and template components:

$$
\begin{array}{l}
\_\ \mathsf{CompMatches}\ \_ : \mathit{TupleComp} \leftrightarrow \mathit{TempComp} \\
\hline
\textbf{let } \mathit{ValueOfTup} == \mathit{TupleValue}^{-1}; \\
\quad \mathit{ValueOfTemp} == \mathit{TempValue}^{-1} \bullet \\
\forall\, \mathit{tupc} : \mathit{TupleComp};\ \mathit{tempc} : \mathit{TempComp} \bullet \\
\quad \mathit{tupc}\ \mathsf{CompMatches}\ \mathit{tempc} \Leftrightarrow \\
\qquad \mathit{tupc} \in \operatorname{ran} \mathit{TupleValue} \land \\
\qquad (\mathit{tempc} \in \operatorname{ran} \mathit{TempValue} \Rightarrow \mathit{ValueOfTup}\ \mathit{tupc} = \mathit{ValueOfTemp}\ \mathit{tempc})
\end{array}
$$

Therefore, only passive tuple components can match, and, if the template component is an actual, the *Value*s in the domain of the corresponding components have to be equal. Only passive tuples are relevant when considering tuple matching; active tuples are invisible to processes. Tuples and templates then match if their arities are equal, their corresponding components match, and the template condition holds. The template condition must yield a Boolean value:

$$
\begin{array}{l}
\_\ \mathsf{Matches}\ \_ : \mathit{APTuple} \leftrightarrow \mathit{Template} \\
\hline
\forall\, \mathit{tup} : \mathit{APTuple};\ \mathit{temp} : \mathit{Template} \bullet \\
\quad \mathit{Type}\,(\mathit{Evaluate}\ \mathit{temp.Condition}) = \mathit{boolean} \land \\
\quad \mathit{tup}\ \mathsf{Matches}\ \mathit{temp} \Leftrightarrow \\
\qquad \mathit{TRUE} = \mathit{Evaluate}\ \mathit{temp.Condition} \land \\
\qquad \mathit{TupArity}(\mathit{tup}) = \mathit{TempArity}(\mathit{temp.List}) \land \\
\qquad (\forall\, i : 1 \,..\, \mathit{TupArity}(\mathit{tup}) \bullet \mathit{tup}(i)\ \mathsf{CompMatches}\ \mathit{temp.List}(i))
\end{array}
$$

CompMatches checks if individual tuple and template components match. Matches checks if tuples and templates match. The exception `type_mismatch` will be raised if the template condition does not yield a Boolean value. This is left in the formal specification.

8.3 Tuple Spaces

Tuple spaces consist of an identity, its specified limit, the tuples it contains, and pending processes in our formal specification. To model pending processes with associated attributes we need an auxiliary type definition for optional statements:

$$\mathit{OptStmt} ::= \mathit{NoStmt} \mid \mathit{IsStmt}\langle\!\langle \mathit{Statement} \rangle\!\rangle$$

We shall also use a predicate for modeling the execution of optional statements based on *Execute*:

$$
\begin{array}{|l}
OptExecute\,\_ : \mathbb{P}\ OptStmt \\
\hline
\textbf{let}\ StatementOf == IsStmt^{-1} \bullet \\
\forall\ os : OptStmt \mid os \in \operatorname{ran} IsStmt \bullet \\
\quad OptExecute\ os \Leftrightarrow Execute\,(StatementOf\ os)
\end{array}
$$

Pending processes are associated with the type of their operation:

$$OpType ::= FetchOp \mid MeetOp \mid MeetIntoOp$$

and with the templates and optional statements as attributes which will later be used for modeling blocking `fetch` and `meet` operations:

$$
\begin{array}{|l}
\hline
\textit{Pending} \\
\hline
proc : Process \\
type : OpType \\
temp : Template \\
os : OptStmt \\
\hline
\end{array}
$$

A tuple space then consists of an identity, its specified limit, a bag of tuples, a bag of pending processes for `fetch` and `meet` operations, and a finite set of processes which are blocked on this full tuple space:

$$
\begin{array}{|l}
\hline
\textit{TupleSpace} \\
\hline
Id : Value \\
Limit : Value \\
Tuples : \operatorname{bag} APTuple \\
PendTemp : \operatorname{bag} Pending \\
PendFull : Process \mathrel{+\!\!\!\!\!\!\mapsto} APTuple \\
\hline
(Type\ Id = atom) \wedge (Id \notin \operatorname{dom} ValuesOfType) \\
(Type\ Limit = integer) \vee (Type\ Limit = om) \\
\mathsf{disjoint}\ \langle \{ pt : \operatorname{dom} PendTemp \bullet pt.proc \}, \operatorname{dom} PendFull \rangle \\
\hline
\end{array}
$$

Bags are defined as follows. bag X is the set of bags of elements of X:

$$\operatorname{bag} X == X \mathrel{+\!\!\!\!\!\rightarrow} \mathbb{N}_1$$

where $X \mathrel{+\!\!\!\!\!\rightarrow} \mathbb{N}_1$ is the set of partial functions from X to $\mathbb{N}_1$. In a bag each element is mapped to the number of times is appears in the bag. $X \mathrel{+\!\!\!\!\!\!\mapsto} Y$ is the set of finite partial functions from X to Y. An indexed family of sets is disjoint if and only if each pair of sets $S(i)$ and $S(j)$ for $i \neq j$ have empty intersection:

$$\mathsf{disjoint}\ S \Leftrightarrow \forall\, i,j : \operatorname{dom} S \mid i \neq j \bullet S(i) \cap S(j) = \{\}$$

Tuple-space identities are atoms (not including the predefined type-atoms). The limit for the number of simultaneously deposited tuples in tuple space has to be an integer or the undefined value. A negative limit is equivalent to 0 (no tuples may be deposited into such a tuple space). The undefined value indicates that no limit has been specified on creation of the tuple space. This limit is a parameter to the function `CreateTS` (Sect. 8.6.1). The main part of a *TupleSpace* it the bag of *APTuple*s. The processes pending for `fetch` and `meet` operations are collected in the bag *PendTemp* of the corresponding *TupleSpace*. The processes pending for `deposit` operations on full *TupleSpace*s are collected in the finite set *PendFull* of the corresponding *TupleSpace*. A process is pending in at most one of these sets (specified by the last property in *TupleSpace*).

Note that we use the terms *pending* and *blocked* as synonyms. A process is *pending* if it has executed a `fetch` or `meet` operation with no matching tuple in tuple space and if no `else` statements are specified (blocking matching). A process is also *pending* for a `deposit` operation on a full tuple space provided that `blockiffull` has been specified and the tuple space is full (see below). Such processes may be reactivated by appropriate events, i.e., by depositing or fetching of tuples.

It would not be sufficient to use sets instead of bags for *PendFetch*: although processes are unique, a tuple-space operation such as the following would yield two identical pending templates for tuple space `TS`:

```
fetch ( 1 )
  xor ( 1 )
   at TS
end fetch;
```

If we would use sets for *PendFetch*, only one such pair would be added to the corresponding set in *TupleSpace*: the second added template would replace the first one. This would not produce real problems, because it does not matter which one of such identical templates might be selected, provided one could be selected at all. However, we have chosen to use bags since this is the precise specification which also does not restrict the design of an implementation unduly (see also Sect. 9.2).

We shall need in the following sections an operation for *schema anti-restriction* for bags of pending processes:

$$\_ \boxbar \_ : \mathbb{F}\, Process \times \mathrm{bag}\, Pending \longrightarrow \mathrm{bag}\, Pending$$

$$\begin{array}{l} \forall\, procs : \mathbb{F}\, Process;\ pends : \mathrm{bag}\, Pending \bullet \\ \quad procs \boxbar pends = \\ \qquad \{\, pe : Pending \mid pe \sqsubseteq pends \wedge pe.proc \notin procs \bullet pe \mapsto pends \sharp pe \,\} \end{array}$$

$\boxed{Z}$ $\mathbb{F}\, S$ is the set of finite subsets of S. The relationship $x \sqsubseteq B$ holds exactly if x appears in bag B a non-zero number of times. The number of times x appears in bag B is $B \sharp x$. $\boxed{Z}$

The schema anti-restriction $PR \boxbar PE$ yields all the members of the bag PE, the *proc* component of which is not a member of the finite set PR of processes. We call $\boxbar$ schema anti-restriction, because of its similarity to the domain anti-restriction $\lhd$ of Z (see below).

Additionally, we shall need the auxiliary function *IDsOF*, which yields the finite set of tuple-space identities from a finite set of tuple spaces:

$$IDsOF : \mathbb{F}\, TupleSpace \longrightarrow \mathbb{F}\, Value$$

$$\begin{array}{l} \forall\, tss : \mathbb{F}\, TupleSpace \bullet \\ \quad IDsOF\ tss = \{\, id : Value \mid (\exists\, ts : tss \bullet ts.Id = id) \,\} \end{array}$$

8.4 Programs and Processes

In this section we will define our program state and the creation and termination of programs and processes. Up to this point in our specification, we only used plain Z for our specification. Now we start with the object-oriented specification of program states. We view the state of a program as the state of a finite set of tuple spaces and a finite set of active processes:

ProgramState

$TSs : \mathbb{F}\ TupleSpace$
$ActiveProcs : \mathbb{F}\ Process$

$$\forall ts1, ts2 : TSs \bullet$$
$$\quad ts1 \neq ts2 \Rightarrow ts1.Id \neq ts2.Id$$
$$\forall ts1, ts2 : TSs \bullet$$
$$\quad \forall tup1, tup2 : \mathrm{dom}(ts1.Tuples \uplus ts2.Tuples);\ i1, i2 : \mathbb{N} \mid$$
$$\qquad tup1(i1) \in \mathrm{ran}\ TupleProcess \land tup2(i2) \in \mathrm{ran}\ TupleProcess \bullet$$
$$\quad (i1 \in \mathrm{dom}\, tup1 \land i2 \in \mathrm{dom}\, tup2 \land tup1(i1) = tup2(i2)) \Rightarrow$$
$$\qquad tup1 = tup2 \land i1 = i2 \land$$
$$\qquad 1 = (ts1.Tuples \uplus ts2.Tuples) \sharp\ tup1$$

Z $\uplus$ denotes the union of bags where the number of times any object appears in the result is the sum of the number of times it appears in the operands. Z

The first property asserts the uniqueness of tuple-space identities and the second property asserts the uniqueness of processes inside active tuples. We now start with the specification of operations on the program state:

INIT

$TSs = \{\}$
$\#ActiveProcs = 1$

A main process is started for the main program on program initialization.

ProgramTermination

$\Delta(TSs, ActiveProcs)$

$TSs' = \{\}$
$ActiveProcs' = \{\}$

Whenever the process for the main program terminates, the entire program terminates (see Sect. 5.2).

For a specification of the entire ProSet language, additional components would be necessary to specify the program state.

Initialization, unlike other operations, can only occur as the first operation and merely determines an initial state (there are no pre-conditions). Semantically, *Init* is interpreted as an operation for obtaining a uniform treatment of histories as sequences of events [Duke *et al.*, 1991].

In principle, we could have defined classes earlier in our specification, for example a class for tuple spaces. We would use object-instances of such a tuple-space class in our program state. But then we could not use the *f*uzz type-checker, which only accepts plain Z specifications [Spivey, 1992a]. Appendix C is a summary of the parts of our specification which were type-checked. This excludes the object-oriented extensions, in particular the class hierarchy, but includes all defined schemas with their formulas. If a type-checker for Object-Z were available, we would have used the object-oriented features of Object-Z more resolutely.

Only processes within *ActiveProcs* are active and executing. Processes, which are suspended for templates or full tuple spaces, are temporally moved to the sets of pending processes of the corresponding tuple spaces. They are reactivated by returning them to *ActiveProcs*. If a process is removed from *ActiveProcs* and moved to a set of pending processes, it is suspended. If, however, a process is removed from *ActiveProcs* and *not* moved to a set of pending processes, then it is terminated (see below). We represent this formally via the class *ActualProcesses*, which contains the external visible variables *ActuallyActiveProcesses*, *ActuallyPendingProcesses*, and *ActuallyExistingProcesses*:

ActualProcesses

$$
\begin{array}{l}
\upharpoonright(ActuallyActiveProcesses, ActuallyPendingProcesses, ActuallyExistingProcesses)\\
ProgramState
\end{array}
$$

$$
\begin{array}{l}
ActuallyActiveProcesses : \mathbb{F}\, Process\\
ActuallyPendingProcesses : \mathbb{F}\, Process\\
ActuallyExistingProcesses : \mathbb{F}\, Process\\
\hline
ActuallyActiveProcesses = ActiveProcs\\
\forall\, p : ActuallyPendingProcesses \bullet\\
\quad (\exists\, ts : TSs \bullet\\
\quad\quad (\exists\, pt : \mathrm{dom}\, ts.PendTemp \bullet p = pt.proc) \lor\\
\quad\quad (\exists\, pf : ts.PendFull \bullet p = \mathit{first}\, pf))\\
\langle ActuallyActiveProcesses, ActuallyPendingProcesses\rangle\\
\qquad\qquad\qquad\qquad \mathsf{partition}\ ActuallyExistingProcesses
\end{array}
$$

$\boxed{Z}$ If p is an ordered pair then $p = (\mathit{first}\, p, \mathit{second}\, p)$ holds. $\boxed{Z}$

We will specify the connection between the computation and the coordination part of ProSet by means of input and output variables of the individual operations. These input and output variables are decorated with ? and !, respectively.

Process creation and termination is specified within the class *ProcessCreationTermination*:

ProcessCreationTermination

$$ProgramState$$

ProcessCreation

$$
\begin{array}{l}
\Delta(ActiveProcs)\\
NewProcess? : Process\\
\hline
ActiveProcs' =\\
\quad ActiveProcs \cup \{NewProcess?\}
\end{array}
$$

Whenever the process creator || is applied in a ProSet program, this process shall be added to the set of active processes.

If a process which has to be removed is part of an active tuple, it has to be replaced by the corresponding return value:

ProcessTermination

$$
\begin{array}{l}
\Delta(TSs, ActiveProcs)\\
ToKill? : Process\\
\hline
TSs' = \{\, ts : TSs;\ ts' : TupleSpace \mid ts.Id = ts'.Id \land\\
\qquad ts'.Limit = ts.Limit \land\\
\qquad ts'.PendTemp = \{ToKill?\} \mathbin{◫} ts.PendTemp \land\\
\qquad ts'.PendFull = \{ToKill?\} \mathbin{⩤} ts.PendFull \land\\
\qquad (\forall\, tup : \mathrm{dom}\, ts.Tuples;\ newtup : APTuple \mid \#tup = \#newtup \land\\
\qquad\qquad\qquad\qquad (\exists\, tupc : \mathrm{ran}\, tup \bullet tupc \in \mathrm{ran}\, TupleProcess) \bullet\\
\qquad\quad (\forall\, i : \mathrm{dom}\, tup \bullet\\
\qquad\qquad newtup(i) = \mathbf{if}\ tup(i) = TupleProcess\ ToKill?\\
\qquad\qquad\qquad\qquad \mathbf{then}\ TupleValue\,(ProcRetVal\ ToKill?)\\
\qquad\qquad\qquad\qquad \mathbf{else}\ tup(i)) \land\\
\qquad\quad ts'.Tuples = (ts.Tuples \mathbin{⩁} \llbracket tup \rrbracket) \uplus \llbracket newtup \rrbracket)\\
\qquad\qquad \bullet\ ts' \,\}\\
ActiveProcs' = ActiveProcs \setminus \{ToKill?\}
\end{array}
$$

$\boxed{Z}$ The domain anti-restriction $S \lhd R$ of a relation R to a set S relates x to y only if R relates x to y and x is not a member of S. We write $[\![a_1, \ldots, a_n]\!]$ for the bag $\{a_1 \mapsto k_1, \ldots, a_n \mapsto k_n\}$ where the elements a_i appear k_i times. E.g., $[\![1, 1]\!] = \{1 \mapsto 2\} = \{(1, 2)\}$. The empty bag is $[\![\,]\!]$. $B \uminus C$ is the bag difference of B and C: the number of times any object appears in it is the number of times it appears in B minus the number of times it appears in C, or zero if that would be negative. $\boxed{Z}$

A terminated process is not only removed from *ActiveProcs*. It is also necessary to remove it from the bags of pending processes in the tuple spaces, and to resolve the future within an active tuple, provided that this process has been spawned as a component of this active tuple. Future resolution is modeled within the above conditional expression of *ProcessTermination*. See Sect. 5.1 for an informal specification for resolving of futures in ProSet. Resolving of futures is only specified for processes within active tuples in tuple space, and not for processes spawned outside of tuple space. This limitation is due to the fact that this specification is not a specification of the entire language.

A newly created process is implicitly added to *ActuallyActiveProcesses* and to *ActuallyExistingProcesses* within the class *ActualProcesses*. Analogously, when a process terminates, it has to be removed from the set of active processes and from the sets of pending processes within *ActualProcesses*, provided it is not the process for the main program. The class *ActualProcesses* is introduced for having a formal specification for actually active, pending, and existing processes. Whenever a process is removed from the set of active processes and is *not* moved to a set of pending processes, this process will be terminated: it is no longer in *ActuallyExistingProcesses*.

8.5 Exceptions

We only indicate exception handling in the present specification through the following generic definition:

‘`escape type_mismatch();`’ : *Statement*
‘`escape ts_invalid_id();`’ : *Statement*
‘`signal ts_is_full();`’ : *Statement*

$\boxed{Z}$ Generic definitions do not have parameters, they uniquely determine the value of the introduced global constant. When no formal generic parameters are supplied, they are also called *unique* axiomatic definitions. We use unique axiomatic definitions to introduce exceptions, because there is nothing more to be said about them in our specification. $\boxed{Z}$

The statement ‘`escape type_mismatch();`’ in ProSet raises the exception `type_mismatch`, which should not be resumed. The exception `ts_invalid_id` will be raised when invalid tuple-space identities are given. The exception `ts_is_full` will be raised when the tuple space requested by a `deposit` operation is full and `blockiffull` has *not* been specified. See Sect. 4.4 for a short informal description of exception handling in ProSet.

8.6 Handling Multiple Tuple Spaces

In this section we define the library functions to handle multiple tuple spaces, as informally specified in Sect. 5.3. The class hierarchy is displayed in Fig. 8.1.

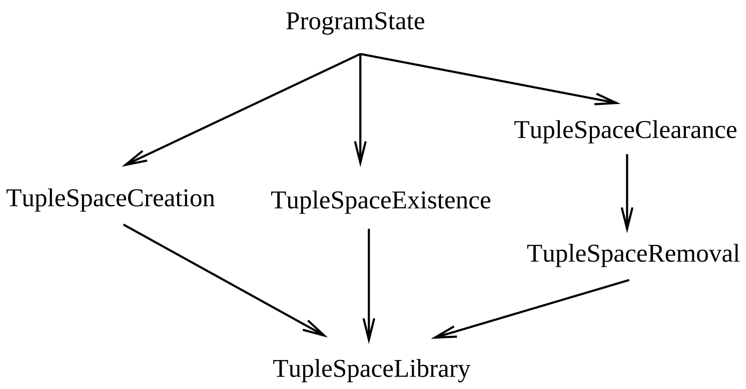


Figure 8.1: The class hierarchy for the library functions to handle multiple tuple spaces.

8.6.1 CreateTS

The library function `CreateTS` creates a new tuple space and returns the corresponding tuple-space identity, provided that the given limit is an integer or the undefined value:

TupleSpaceCreation

$\upharpoonright(CreateTS)$

$ProgramState$

CreateTSok

$$
\begin{array}{l}
\Delta(TSs)\\
InLimit? : Value\\
Return! : Value\\
\hline
Type\ InLimit? = integer \lor\\
Type\ InLimit? = om\\
\exists_1\ a : Value;\ ts : TupleSpace \mid\\
\qquad Type\ a = atom \land\\
\qquad a \notin IDsOF\ TSs \land\\
\qquad a \notin \operatorname{dom} ValuesOfType \land\\
\qquad ts.Id = a \land\\
\qquad ts.Limit = InLimit? \land\\
\qquad ts.Tuples = [\![\,]\!] \land\\
\qquad ts.PendTemp = [\![\,]\!] \land\\
\qquad ts.PendFull = \{\} \bullet\\
\quad TSs' = TSs \cup \{ts\} \land\\
\quad Return! = a
\end{array}
$$

$\boxed{Z}$ The predicate $\exists_1 S \bullet P$ is true if there is exactly one way of giving values to the bound variables introduced by S so that both the property S and the predicate P are true. $\boxed{Z}$

A new, empty tuple space is created this way. The tuple-space identity (an atom) will be created via a call to the standard library function `newat`, which returns a new, unique atom (Sect. 4.1). For our specification it is sufficient to specify that this tuple-space identity is unique with respect to our program state. The new tuple-space identity will be returned.

The limit for the number of simultaneously deposited tuples in tuple space has to be an integer or the undefined value:

$$
\begin{array}{|l}
\textit{CreateTSTypeMismatch} \\
\hline
\textit{InLimit}? : \textit{Value} \\
\textit{Exception}! : \textit{Statement} \\
\hline
\neg\,(\textit{Type InLimit}? = \textit{integer} \lor \textit{Type InLimit}? = \textit{om}) \\
\textit{Exception}! = \text{`}\texttt{escape type\_mismatch();}\text{'} \\
\hline
\end{array}
$$

$\textit{CreateTS} \mathrel{\widehat{=}} \textit{CreateTSok} \lor \textit{CreateTSTypeMismatch}$

This last definition introduces a new schema called *CreateTS*, obtained by combining the two schemas on the right-hand side via disjunction.

Exception handling is specified for operations which may change the state in separate schemas without Δ-lists to emphasize that the state does not change when such an error occurs. As noted in Sect. B.6, not indicating a Δ-list is equivalent to specifying an empty Δ-list. Therefore, operations with an empty Δ-lists do not change the state.

$\boxed{Z}$ The operation *CreateTS* could be specified directly by writing a single schema which combines the predicate parts of the two schemas *CreateTSok* and *CreateTSTypeMismatch*. The effect of the schema ∨ operator is to make a schema in which the predicate part is the result of joining the predicate parts of its arguments with the logical connective ∨. Similarly, the effect of the schema ∧ operator is to take the conjunction of the two predicate parts. Any common variables of the two schemas are merged. We sketch an alternative specification of *CreateTS*:

$$
\begin{array}{|l}
\textit{CreateTS} \\
\hline
\Delta(\textit{TSs}) \\
\textit{InLimit}? : \textit{Value} \\
\textit{Return}! : \textit{Value} \\
\textit{Exception}! : \textit{Statement} \\
\hline
((\textit{Type InLimit}? = \textit{integer} \lor \textit{Type InLimit}? = \textit{om}) \land \\
\quad \exists_1\, a : \textit{Value};\ ts : \textit{TupleSpace} \mid \\
\qquad \ldots) \\
\qquad\quad \lor \\
(\neg\,(\textit{Type InLimit}? = \textit{integer} \lor \textit{Type InLimit}? = \textit{om}) \land \\
\quad \textit{Exception}! = \text{`}\texttt{escape type\_mismatch();}\text{'} \land \\
\quad \textit{TSs}' = \textit{TSs}) \\
\hline
\end{array}
$$

In order to write *CreateTS* as a single schema, it has been necessary to write out explicitly that the state of *TSs* does not change when an exception is raised. $\boxed{Z}$

We do not specify the return values when exceptions are raised.

8.6.2 ExistsTS

The library function `ExistsTS` checks whether a given atom is a valid tuple-space identity:

$$
\begin{array}{l}
\textit{TupleSpaceExistence} \\
\hline
\upharpoonright(\textit{ExistsTS}) \\
\textit{ProgramState} \\
\quad \textit{ExistsTS} \\
\quad \hline
\quad \textit{InTS?} : \textit{Value} \\
\quad \textit{Return!} : \textit{Value} \\
\quad \textit{Exception!} : \textit{Statement} \\
\quad \hline
\quad (\textit{Type InTS?} = \textit{atom} \land \textit{Return!} = \mathbf{if}\ \textit{InTS?} \in \textit{IDsOF TSs} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ \textit{TRUE} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \textit{FALSE}) \\
\qquad \lor \\
\quad (\textit{Type InTS?} \neq \textit{atom} \land \textit{Exception!} = \text{`}\texttt{escape type\_mismatch();}\text{'})
\end{array}
$$

ExistsTS does not modify the program state. It returns a Boolean *Value* indicating the existence of the given tuple-space identity, which has to be an atom.

8.6.3 ClearTS

The library function `ClearTS` removes all active and passive tuples from a specified tuple space:

$$
\begin{array}{l}
\textit{TupleSpaceClearance} \\
\hline
\upharpoonright(\textit{ClearTS}) \\
\textit{ProgramState} \\
\quad \textit{ClearTSok} \\
\quad \hline
\quad \Delta(\textit{TSs}, \textit{ActiveProcs}) \\
\quad \textit{InTS?} : \textit{Value} \\
\quad \textit{Return!} : \textit{Value} \\
\quad \hline
\quad \textit{InTS?} \in \textit{IDsOF TSs} \\
\quad \mathbf{let}\ \textit{ClearProcs} == \{\, ts : \textit{TSs};\ p : \textit{ActiveProcs} \mid ts.\textit{Id} = \textit{InTS?} \land \\
\qquad\qquad (\exists\, tup : \operatorname{dom} ts.\textit{Tuples} \bullet \exists\, tupc : \operatorname{ran} tup \bullet tupc = \textit{TupleProcess}\ p) \bullet p \,\} \bullet \\
\quad \textit{TSs}' = \{\, ts : \textit{TSs};\ ts' : \textit{TupleSpace} \mid ts.\textit{Id} = ts'.\textit{Id} \land \\
\qquad\quad ts'.\textit{Limit} = ts.\textit{Limit} \land \\
\qquad\quad ts'.\textit{Tuples} = \mathbf{if}\ ts'.\textit{Id} = \textit{InTS?} \\
\qquad\qquad\qquad\qquad \mathbf{then}\ [\![\,]\!] \\
\qquad\qquad\qquad\qquad \mathbf{else}\ ts.\textit{Tuples} \land \\
\qquad\quad ts'.\textit{PendTemp} = \textit{ClearProcs} \mathbin{◫} ts.\textit{PendTemp} \land \\
\qquad\quad ts'.\textit{PendFull} = \textit{ClearProcs} \mathbin{⩤} ts.\textit{PendFull} \bullet ts' \,\} \land \\
\quad \textit{ActiveProcs}' = \textit{ActiveProcs} \setminus \textit{ClearProcs} \\
\quad \textit{Return!} = \textit{om} \\
\\
\quad \textit{ClearTSinvalid} \\
\quad \hline
\quad \textit{InTS?} : \textit{Value} \\
\quad \textit{Exception!} : \textit{Statement} \\
\quad \hline
\quad \textit{InTS?} \notin \textit{IDsOF TSs} \\
\quad \textit{Exception!} = \mathbf{if}\ \textit{Type InTS?} = \textit{atom} \\
\qquad\qquad\qquad \mathbf{then}\ \text{`}\texttt{escape ts\_invalid\_id();}\text{'} \\
\qquad\qquad\qquad \mathbf{else}\ \text{`}\texttt{escape type\_mismatch();}\text{'}
\end{array}
$$

$$ClearTS \mathrel{\widehat=} ClearTSok \lor ClearTSinvalid$$

The processes within removed active tuples (the set *ClearProcs*) are also removed from *ActiveProcs* and the bags of pending processes, and thus have their execution terminated. Note that the processes which are not in *ClearProcs* are *not* removed from the bags of pending processes of *InTS*?. Only the bag *Tuples* is cleared in *InTS*?. The return value is the undefined value in any case.

The exception `type_mismatch` and not `ts_invalid_id` will be raised if the actual parameter for `ClearTS` is not an atom.

8.6.4 RemoveTS

The library function `RemoveTS` calls `ClearTS` and removes the given tuple space from the set of tuple spaces:

TupleSpaceRemoval

$\upharpoonright(RemoveTS)$

$TupleSpaceClearance$ [**remove** $ClearTSok, ClearTSinvalid$]

RemoveTSfromState

$\Delta(TSs)$
$InTS? : Value$

$TSs' = TSs \setminus \{\, ts : TSs \mid ts.Id = InTS? \,\}$

The auxiliary operation *RemoveTSfromState* removes the entire tuple space from the program state. This may terminate pending processes, which are blocked on *InTS*?. *RemoveTS* is the sequential composition of *ClearTS* and *RemoveTSfromState*:

$$RemoveTS \mathrel{\widehat=} ClearTS \mathbin{⨟} RemoveTSfromState$$

$\boxed{Z}$ If $Op1$ and $Op2$ are schemas describing two operations, then $Op1 ⨟ Op2$ is a schema which describes their sequential composition. The components of $Op1 ⨟ Op2$ are the undecorated components of $Op1$ and the primed components of $Op2$, together with their merged inputs and outputs. Conversely, the components of the piping $Op1 \gg Op2$ are the inputs of $Op1$ and the outputs of $Op2$, together with their merged decorated and undecorated components (the outputs of $Op1$ have to match the inputs of $Op2$ in this case). $\boxed{Z}$

Auxiliary features are removed with the keyword **remove** while inheriting a class as we indicate above with *ClearTSok* and *ClearTSinvalid*. Unfortunately, it is not possible in Object-Z to indicate which features of a class are available to its children (users through inheritance). See also Sect. B.7 for a discussion on this subject.

8.6.5 The Tuple-Space Library

The tuple-space library for handling multiple tuple spaces is now defined as follows:

TupleSpaceLibrary

$\upharpoonright(CreateTS, ExistsTS, ClearTS, RemoveTS)$

$TupleSpaceCreation$ [**remove** $CreateTSok, CreateTSTypeMismatch$]
$TupleSpaceExistence$
$TupleSpaceRemoval$ [**remove** $RemoveTSfromState$]

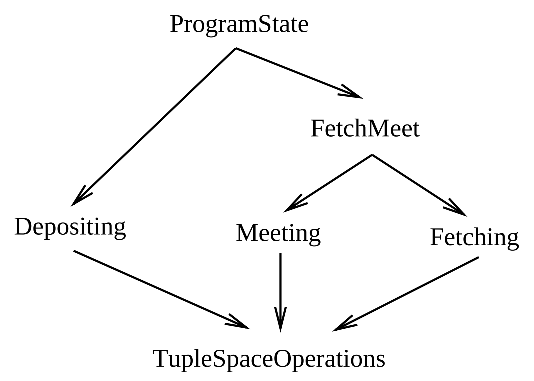


Figure 8.2: The class hierarchy for the tuple-space operations.

Whenever objects of this class are instantiated, only the four library functions are available to such instances of the *TupleSpaceLibrary*. Note that children of this class still have access to the state variables of *ProgramState*, but not to the removed auxiliary operations.

8.7 Tuple-Space Operations

A concise overview of the abstract grammar for the tuple-space operations is presented in Appendix A using BNF (Backus Naur Form). Conversely, the informal semantics in Chap. 5 is presented together with syntax diagrams, which are spread over the text. For a concise overview, we consider BNF as more appropriate.

Section 8.7.1 will provide some preliminary definitions for the tuple-space operations. The *Deposit* operation will be defined in Sect. 8.7.2. Sections 8.7.3 and 8.7.4 will define the *Fetch* and *Meet* operations, respectively. The class hierarchy for the tuple-space operations is displayed in Fig. 8.2.

8.7.1 Some Preliminary Definitions

Because of Z's principle of *definition before use*, we have to introduce the definitions of this subsection before we use them in the following subsections, which define the tuple-space operations. See also Sect. B.7 for a discussion on this subject.

We will use the predicate *HasIntos* for the pending processes of tuple spaces, which checks if there are `into` expressions associated within a template:

$$
\begin{array}{|l}
HasIntos\_ : \mathbb{P}\, Template \\
\hline
\textbf{let}\ FormalOf == TempFormal^{-1} \bullet \\
\forall pl : Pending \bullet \\
\quad HasIntos\ pl.temp \Leftrightarrow \\
\qquad (\exists tempc : \operatorname{ran} pl.temp.List \mid tempc \in \operatorname{ran} TempFormal \bullet \\
\qquad\quad (FormalOf\ tempc).Into \neq NoInto)
\end{array}
$$

We need an auxiliary function that yields the number of elements in a bag to control the limits of tuple spaces:

$$
\begin{array}{l}
[X] \\
\hline
\mathit{BagSum} : \mathrm{bag}\, X \longrightarrow \mathbb{N} \\
\hline
\forall\, b : \mathrm{bag}\, X;\ x : X \bullet \\
\qquad \mathit{BagSum}\, [\![\,]\!] = 0 \land \\
\qquad \mathit{BagSum}\, ([\![x]\!] \uplus b) = 1 + \mathit{BagSum}\, b
\end{array}
$$

A function that yields the $\mathbb{Z}$-integer value from a *Value* with type *integer*:

$$
\begin{array}{l}
\mathit{IntValueOf} : \mathit{Value} \longrightarrow \mathbb{Z} \\
\hline
\forall\, i : \mathit{Value} \mid \mathit{Type}\ i = \mathit{integer} \bullet \\
\qquad \exists\, z : \mathbb{Z} \bullet \mathit{IntValueOf}\ i = z
\end{array}
$$

The *BlockMode* indicates whether `blockiffull` has been specified:

$$\mathit{BlockMode} ::= \mathit{BlockIfFull} \mid \mathit{DoNotBlock}$$

The structure of the input for the *Fetch* and *Meet* operations corresponds to their syntactical structure:

$$\mathit{TempList} == \mathrm{seq}_1(\mathit{Template} \times \mathit{OptStmt}) \times \mathit{Value}$$

$\boxed{Z}$ $\mathrm{seq}_1\, X = \mathrm{seq}\, X \setminus \{\langle\rangle\}$ $\boxed{Z}$

The function *MakePends* creates from such a *TempList*, a process, and the operation's type a corresponding bag of pending processes:

$$
\begin{array}{l}
\mathit{MakePends} : \mathit{TempList} \times \mathit{Process} \times \mathit{OpType} \longrightarrow \mathrm{bag}\, \mathit{Pending} \\
\hline
\forall\, \mathit{tl} : \mathit{TempList};\ \mathit{tos} : \mathit{Template} \times \mathit{OptStmt};\ \mathit{tosl} : \mathrm{seq}(\mathit{Template} \times \mathit{OptStmt}); \\
\quad \mathit{id} : \mathit{Value};\ \mathit{pr} : \mathit{Process};\ \mathit{ot} : \mathit{OpType} \bullet \\
\qquad \mathit{MakePends}((\langle\rangle, \mathit{id}), \mathit{pr}, \mathit{ot}) = [\![\,]\!] \land \\
\qquad \mathit{MakePends}((\langle \mathit{tos} \rangle \frown \mathit{tosl}, \mathit{id}), \mathit{pr}, \mathit{ot}) = \mathit{MakePends}((\mathit{tosl}, \mathit{id}), \mathit{pr}, \mathit{ot}) \uplus \\
\qquad\qquad \{\, \mathit{pe} : \mathit{Pending} \mid \mathit{pe.proc} = \mathit{pr} \land \mathit{pe.temp} = \mathit{first}\ \mathit{tos} \land \mathit{pe.os} = \mathit{second}\ \mathit{tos} \land \\
\qquad\qquad\qquad\qquad \mathit{pe.type} = \mathbf{if}\ \mathit{HasIntos}(\mathit{first}\ \mathit{tos}) \\
\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ \mathit{MeetIntoOp} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \mathit{ot} \qquad\qquad \bullet\ \mathit{pe} \mapsto 1 \,\}
\end{array}
$$

$\boxed{Z}$ $\frown$ denotes the concatenation of sequences. $\boxed{Z}$

We will need some auxiliary projection functions for Z-tuples with three components:

$$
\begin{array}{l}
[A, B, C] \\
\hline
\mathit{GetTemp} : A \times B \times C \longrightarrow A \\
\mathit{GetTup} : A \times B \times C \longrightarrow B \\
\mathit{GetOS} : A \times B \times C \longrightarrow C \\
\hline
\forall\, a : A;\ b : B;\ c : C \bullet \\
\qquad \mathit{GetTemp}(a, b, c) = a \land \\
\qquad \mathit{GetTup}(a, b, c) = b \land \\
\qquad \mathit{GetOS}(a, b, c) = c
\end{array}
$$

As a first step for depositing tuples we define the recursive function AddTuple for adding a single tuple to a specified tuple space of a program state:

$$
\begin{array}{l}
\,\mathsf{AddTuple}\, : (\mathbb{F}\ \mathit{TupleSpace} \times \mathbb{F}\ \mathit{Process}) \times (\mathit{APTuple} \times \mathit{Value}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \longrightarrow (\mathbb{F}\ \mathit{TupleSpace} \times \mathbb{F}\ \mathit{Process}) \\
\hline
\forall\, \mathit{tss}, \mathit{tss}' : \mathbb{F}\ \mathit{TupleSpace};\ \mathit{AP}, \mathit{AP}' : \mathbb{F}\ \mathit{Process};\ \mathit{tup} : \mathit{APTuple};\ \mathit{id} : \mathit{Value} \bullet \\
\quad (\mathit{tss}', \mathit{AP}') = (\mathit{tss}, \mathit{AP})\ \mathsf{AddTuple}\ (\mathit{tup}, \mathit{id}) \Leftrightarrow \\
\qquad (\exists_1\, \mathit{myts} : \mathit{tss} \mid \mathit{myts}.\mathit{Id} = \mathit{id} \bullet \\
\qquad (\mathbf{let}\ \mathit{MatchMeets} == \{\, \mathit{pt} : \mathrm{dom}\, \mathit{myts}.\mathit{PendTemp} \mid \mathit{tup}\ \mathsf{Matches}\ \mathit{pt}.\mathit{temp} \wedge \\
\qquad\qquad\qquad\qquad\qquad \mathit{pt}.\mathit{type} = \mathit{MeetOp} \,\}; \\
\qquad\quad \mathit{MatchFetchs} == \{\, \mathit{pt} : \mathrm{dom}\, \mathit{myts}.\mathit{PendTemp} \mid \mathit{tup}\ \mathsf{Matches}\ \mathit{pt}.\mathit{temp} \wedge \\
\qquad\qquad\qquad\qquad\qquad \mathit{pt}.\mathit{type} = \mathit{FetchOp} \,\}; \\
\qquad\quad \mathit{NewProcs} == \{\, p : \mathit{Process} \mid (\exists\, \mathit{tupc} : \mathrm{ran}\, \mathit{tup} \bullet \mathit{tupc} = \mathit{TupleProcess}\, p) \,\} \bullet \\
\qquad (\mathbf{let}\ \mathit{MatchIntos} == \{\, \mathit{pt} : \mathrm{dom}\, \mathit{myts}.\mathit{PendTemp} \mid \mathit{tup}\ \mathsf{Matches}\ \mathit{pt}.\mathit{temp} \wedge \\
\qquad\qquad\qquad\qquad\qquad \mathit{pt}.\mathit{type} = \mathit{MeetIntoOp} \wedge \\
\qquad\qquad\qquad\qquad\qquad \neg\, (\exists\, \mathit{pm} : \mathit{MatchMeets} \bullet \mathit{pm}.\mathit{proc} = \mathit{pt}.\mathit{proc}) \,\} \bullet \\
\qquad (\forall\, \mathit{pm} : \mathbb{P}\ \mathit{MatchMeets} \mid (\forall\, p1, p2 : \mathit{pm} \bullet p1.\mathit{proc} = p2.\mathit{proc}) \bullet \exists_1\, \mathit{pend} : \mathit{pm} \bullet \\
\qquad\quad \mathit{pend}.\mathit{temp}\ \mathsf{FormalAssign}\ \mathit{tup} \wedge \\
\qquad\quad \mathit{OptExecute}\, \mathit{pend}.\mathit{os}) \wedge \\
\qquad ((\mathit{MatchIntos} \cup \mathit{MatchFetchs} = \{\}) \Rightarrow \\
\qquad\quad (\mathit{tss}' = \{\, \mathit{ts} : \mathit{tss};\ \mathit{ts}' : \mathit{TupleSpace} \mid \mathit{ts}'.\mathit{Id} = \mathit{ts}.\mathit{Id} \wedge \\
\qquad\qquad\qquad \mathit{ts}'.\mathit{Limit} = \mathit{ts}.\mathit{Limit} \wedge \\
\qquad\qquad\qquad \mathit{ts}'.\mathit{Tuples} = \mathbf{if}\ \mathit{ts}'.\mathit{Id} = \mathit{id} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ \mathit{ts}.\mathit{Tuples} \uplus \llbracket \mathit{tup} \rrbracket \\
\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \mathit{ts}.\mathit{Tuples} \wedge \\
\qquad\qquad\qquad \mathit{ts}'.\mathit{PendTemp} = \\
\qquad\qquad\qquad\quad \{\, \mathit{mm} : \mathit{MatchMeets} \bullet \mathit{mm}.\mathit{proc} \,\} \mathbin{◫} \mathit{ts}.\mathit{PendTemp} \wedge \\
\qquad\qquad\qquad \mathit{ts}'.\mathit{PendFull} = \mathit{ts}.\mathit{PendFull} \bullet \mathit{ts}' \,\} \wedge \\
\qquad\quad \mathit{AP}' = \mathit{AP} \cup \mathit{NewProcs} \cup \{\, \mathit{mm} : \mathit{MatchMeets} \bullet \mathit{mm}.\mathit{proc} \,\})) \wedge \\
\qquad ((\mathit{MatchIntos} \cup \mathit{MatchFetchs} \neq \{\}) \Rightarrow \\
\qquad (\exists\, \mathit{mif} : (\mathit{MatchIntos} \cup \mathit{MatchFetchs}) \bullet \\
\qquad\quad \mathit{mif}.\mathit{temp}\ \mathsf{FormalAssign}\ \mathit{tup} \wedge \\
\qquad\quad \mathit{OptExecute}\, \mathit{mif}.\mathit{os} \wedge \\
\qquad\quad (\mathit{mif} \in \mathit{MatchFetchs} \Rightarrow \\
\qquad\qquad (\mathit{tss}' = \{\, \mathit{ts} : \mathit{tss};\ \mathit{ts}' : \mathit{TupleSpace} \mid \mathit{ts}'.\mathit{Id} = \mathit{ts}.\mathit{Id} \wedge \\
\qquad\qquad\qquad\quad \mathit{ts}'.\mathit{Limit} = \mathit{ts}.\mathit{Limit} \wedge \\
\qquad\qquad\qquad\quad \mathit{ts}'.\mathit{Tuples} = \mathit{ts}.\mathit{Tuples} \wedge \\
\qquad\qquad\qquad\quad \mathit{ts}'.\mathit{PendTemp} = (\{\, \mathit{mm} : \mathit{MatchMeets} \bullet \mathit{mm}.\mathit{proc} \,\} \cup \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{\mathit{mif}.\mathit{proc}\}) \mathbin{◫} \mathit{ts}.\mathit{PendTemp} \wedge \\
\qquad\qquad\qquad\quad \mathit{ts}'.\mathit{PendFull} = \mathit{ts}.\mathit{PendFull} \bullet \mathit{ts}' \,\} \wedge \\
\qquad\qquad \mathit{AP}' = \mathit{AP} \cup \{\, \mathit{mm} : \mathit{MatchMeets} \bullet \mathit{mm}.\mathit{proc} \,\} \cup \{\mathit{mif}.\mathit{proc}\})) \wedge \\
\qquad\quad (\mathit{mif} \in \mathit{MatchIntos} \Rightarrow \\
\qquad\quad (\exists_1\, \mathit{newTSs} : \mathbb{F}\ \mathit{TupleSpace};\ \mathit{newAP} : \mathbb{F}\ \mathit{Process} \bullet \\
\qquad\qquad \mathit{newTSs} = \{\, \mathit{ts} : \mathit{tss};\ \mathit{newts} : \mathit{TupleSpace} \mid \mathit{newts}.\mathit{Id} = \mathit{ts}.\mathit{Id} \wedge \\
\qquad\qquad\qquad\quad \mathit{newts}.\mathit{Limit} = \mathit{ts}.\mathit{Limit} \wedge \\
\qquad\qquad\qquad\quad \mathit{newts}.\mathit{Tuples} = \mathit{ts}.\mathit{Tuples} \wedge \\
\qquad\qquad\qquad\quad \mathit{newts}.\mathit{PendTemp} = (\{\, \mathit{mm} : \mathit{MatchMeets} \bullet \mathit{mm}.\mathit{proc} \,\} \cup \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{\mathit{mif}.\mathit{proc}\}) \mathbin{◫} \mathit{ts}.\mathit{PendTemp} \wedge \\
\qquad\qquad\qquad\quad \mathit{newts}.\mathit{PendFull} = \mathit{ts}.\mathit{PendFull} \bullet \mathit{newts} \,\} \wedge \\
\qquad\qquad \mathit{newAP} = \mathit{AP} \cup \{\, \mathit{mm} : \mathit{MatchMeets} \bullet \mathit{mm}.\mathit{proc} \,\} \cup \{\mathit{mif}.\mathit{proc}\} \wedge \\
\qquad\qquad (\mathit{tss}', \mathit{AP}') = \\
\qquad\qquad\quad (\mathit{newTSs}, \mathit{newAP})\ \mathsf{AddTuple}\ (\mathit{mif}.\mathit{temp}\ \mathsf{EvalIntos}\ \mathit{tup}, \mathit{id})))))))
\end{array}
$$

Since this is quite a longish formula we give an informal outline of the predicate on the right-hand side of $\Leftrightarrow$ in the previous schema. The local set *MatchMeets* within the **let** declaration contains all the pending processes which are pending for `meet` operations without `into` expressions, where the associated templates match the tuple to be added. *MatchIntos* contains all the pending processes which are pending for `meet` operations with `into` expressions, where the associated templates match the tuple to be added. Note that no process is contained in both, *MatchMeets* and *MatchIntos*. *MatchFetchs* contains all the pending processes which are pending for `fetch` operations, where the associated templates match the tuple to be added. *NewProcs* contains the processes which are active within the tuple to be added. *NewProcs* is empty if the tuple to be added matches any template in *tss*, because only passive tuples can match. An informal outline of the **let** predicate follows:

```
(for all meets without intos that match (but for each process only once):
    (assign the tuple fields to the l-values in the corresponding formals (optional)) ∧
    (execute the associated statements (optional))
            ∧
(if not exists a meet with intos or a fetch, which matches:
    add the tuple))                                                  (1)
            ∧
 (if exists a meet with intos or a fetch, which matches:
    (assign the tuple fields to the l-values in the corresponding formals (optional)) ∧
    (execute the associated statements (optional)) ∧
    (if a fetch was selected:
                                satisfy the selected fetch) ∧        (2)
    (if a meet with intos was selected:
                                satisfy the selected meet))          (3)
```

We expand the three numbered operation parts of this schema. Adding a tuple for which no pending `meet` with `into`s and no pending `fetch` exists (1):

(add the tuple to the the bag-component *Tuples*) ∧
(remove the processes which were pending for `meet`s without `into`s from the list of pending processes (*PendTemp*)) ∧
(add the contained active processes to *ActiveProcs*) ∧
(reactivate the processes which were pending for `meet`s without `into`s)

Satisfaction of a `fetch` which matches implies the following actions (2):

(remove the processes which were pending for `meet`s without `into`s from the list of pending processes (*PendTemp*)) ∧
(remove the associated process from the list of pending processes (*PendTemp*)) ∧
(reactivate the processes which were pending for `meet`s without `into`s) ∧
(reactivate the associated process)

If we satisfy a matching, pending `fetch`, we do not add the tuple. Satisfaction of a `meet` with `into`s, which matches implies the following actions (3):

(remove the processes which were pending for `meet`s without `into`s from the `meet`-lists of pending processes) ∧
(remove the associated process from the list of pending processes (*PendTemp*)) ∧
(reactivate the processes which were pending for `meet`s without `into`s) ∧
(reactivate the associated process) ∧
(add the changed tuple recursively with AddTuple)

For a matching `meet` template with `into`s the changed tuple and not the original tuple has to be added. This may satisfy other pending templates, and therefore AddTuple may call itself recursively an arbitrary but finite number of times.

8.7.2 Depositing Tuples

This section defines the *Deposit* operation. The informal specification is given in Sect. 5.4.1.

Depositing

$\upharpoonright (Deposit)$

$ProgramState$

DepositOK

$\Delta(TSs, ActiveProcs)$
$ToDeposit? : APTuple \times Value$

$second\ ToDeposit? \in IDsOF\ TSs$

$(TSs', ActiveProcs') = (TSs, ActiveProcs)\ \mathsf{AddTuple}\ \ ToDeposit?$

DepositInvalid

$ToDeposit? : APTuple \times Value$
$Exception! : Statement$

$second\ ToDeposit? \notin IDsOF\ TSs$

$Exception! =$ '`escape ts_invalid_id();`'

Blocking on full tuple spaces (provided that `blockiffull` has been specified):

FullTSBlock

$\Delta(TSs, ActiveProcs)$
$ToDeposit? : APTuple \times Value$
$InProc? : Process$
$Blocking? : BlockMode$

$second\ ToDeposit? \in IDsOF\ TSs$

$Blocking? = BlockIfFull$

$$\begin{aligned}TSs' = \{\, & ts : TSs;\ ts' : TupleSpace \mid ts'.Id = ts.Id \land\\ & ts'.Limit = ts.Limit \land\\ & ts'.Tuples = ts.Tuples \land\\ & ts'.PendTemp = ts.PendTemp \land\\ & ts'.PendFull = \mathbf{if}\ ts'.Id = second\ ToDeposit?\\ & \qquad \mathbf{then}\ ts.PendFull \cup \{\, InProc? \mapsto first\ ToDeposit? \,\}\\ & \qquad \mathbf{else}\ ts.PendFull \bullet ts' \,\}\end{aligned}$$

$ActiveProcs' = ActiveProcs \setminus \{InProc?\}$

FullTSException

$ToDeposit? : APTuple \times Value$
$Blocking? : BlockMode$
$Exception! : Statement$

$second\ ToDeposit? \in IDsOF\ TSs$

$Blocking? = DoNotBlock$

$Exception! =$ '`signal ts_is_full();`'

The exception `ts_is_full` will be raised when the requested tuple space is full and `blockiffull` has *not* been specified.

Note that `ts_invalid_id` will be raised if both exceptional conditions — invalid tuple-space identity and full tuple space — hold.

TSisFull

$ToDeposit? : APTuple \times Value$

$\exists\, ts : TSs \mid ts.Id = second\ ToDeposit? \bullet$
$\quad Type\ ts.Limit = integer \wedge$
$\quad IntValueOf\ ts.Limit \leq BagSum\ ts.Tuples$

$$Deposit \mathrel{\widehat=} (DepositOK \wedge \neg\, TSisFull) \vee ((FullTSException \vee FullTSBlock) \wedge TSisFull) \vee DepositInvalid$$

We only indicate exception handling for full tuple spaces as informally described in Sect. 5.4.1. It is not specified that the exception `type_mismatch` will be raised if the tuple-operands are not tuples. Since *Deposit* deposits passive and active tuples, the input-tuples cannot be first-class ProSet-values: we need objects of type *APTuple*.

8.7.3 Fetching Tuples

This section defines the *Fetch* operation. The informal specification is given in Sect. 5.4.2. When `else` statements are to be executed or invalid template lists are given, the operations for fetching and meeting behave in the same way:

FetchMeet

ProgramState

If no matching tuple has been found and `else` statements are specified, our program state does not change, and the `else` statements are executed:

DoElseStmt

$InTempList? : TempList$
$InProc? : Process$
$Else? : OptStmt$

$second\ InTempList? \in IDsOF\ TSs$

$\neg\, (\exists\, ts : TSs \mid (ts.Id = second\ InTempList?) \bullet$
$\quad \exists\, tos : \operatorname{ran}(first\ InTempList?);\ tup : \operatorname{dom} ts.Tuples \bullet tup\ \mathsf{Matches}\ (first\ tos))$

$Else? \neq NoStmt$

$OptExecute\ Else?$

The exception `ts_invalid_id` will be raised when an invalid tuple-space identity is given within the template list:

InvalidTempList

$InTempList? : TempList$
$InProc? : Process$
$Exception! : Statement$

$second\ InTempList? \notin IDsOF\ TSs$

$Exception! =$ ‘`escape ts_invalid_id();`’

Fetching of a tuple is then specified in the class *Fetching* based on *FetchMeet*:

```
Fetching
$\upharpoonright (Fetch)$
$FetchMeet$

  FetchMatch
  $\Delta(TSs, ActiveProcs)$
  $InTempList? : TempList$
  $InProc? : Process$
  ----------------
  $second\ InTempList? \in IDsOF\ TSs$
  $\exists\, ts : TSs \mid (ts.Id = second\ InTempList?) \bullet$
      $\exists\, tos : \mathrm{ran}(first\ InTempList?);\ tup : \mathrm{dom}\, ts.Tuples \bullet tup\ \mathsf{Matches}\ (first\ tos)$
  $\exists\, myts : TSs \mid (myts.Id = second\ InTempList?) \bullet$
  $\mathbf{let}\ Matchings == \{\, TEMP : Template;\ TUP : APTuple;\ OS : OptStmt \mid$
                              $(\exists\, tos : \mathrm{ran}(first\ InTempList?);\ tup : \mathrm{dom}\, myts.Tuples \bullet$
                                  $tup\ \mathsf{Matches}\ (first\ tos) \wedge$
                                  $TEMP = first\ tos \wedge TUP = tup \wedge OS = second\ tos) \bullet$
                              $(TEMP, TUP, OS)\,\} \bullet$
  $(\exists_1\, SelMatch : Matchings \bullet$
      $(GetTemp\ SelMatch)\ \mathsf{FormalAssign}\ (GetTup\ SelMatch) \wedge$
      $OptExecute\ (GetOS\ SelMatch) \wedge$
      $((myts.PendFull = \{\}) \Rightarrow$
          $(TSs' = \{\, ts : TSs;\ ts' : TupleSpace \mid ts.Id = ts'.Id \wedge$
                  $ts'.Limit = ts.Limit \wedge$
                  $ts'.Tuples = \mathbf{if}\ myts.Id = ts.Id$
                                  $\mathbf{then}\ ts.Tuples \uplus [\![GetTup\ SelMatch]\!]$
                                  $\mathbf{else}\ ts.Tuples \wedge$
                  $ts'.PendTemp = ts.PendTemp \wedge$
                  $ts'.PendFull = ts.PendFull \bullet ts'\,\} \wedge$
          $ActiveProcs' = ActiveProcs)) \wedge$
      $((myts.PendFull \neq \{\}) \Rightarrow$
      $(\exists_1\, SelBlocked : myts.PendFull \bullet$
          $(TSs' = \{\, ts : TSs;\ ts' : TupleSpace \mid ts.Id = ts'.Id \wedge$
                  $ts'.Limit = ts.Limit \wedge$
                  $ts'.Tuples = \mathbf{if}\ myts.Id = ts.Id$
                                  $\mathbf{then}\ (ts.Tuples \uplus [\![GetTup\ SelMatch]\!])$
                                          $\uplus [\![second\ SelBlocked]\!]$
                                  $\mathbf{else}\ ts.Tuples \wedge$
                  $ts'.PendTemp = ts.PendTemp \wedge$
                  $ts'.PendFull = \mathbf{if}\ myts.Id = ts.Id$
                                    $\mathbf{then}\ ts.PendFull \setminus \{SelBlocked\}$
                                    $\mathbf{else}\ ts.PendFull \bullet ts'\,\} \wedge$
          $ActiveProcs' = ActiveProcs \cup \{first\ SelBlocked\}))))$

  No into expressions are allowed for fetch operations:

  DisallowIntos
  $InTempList? : TempList$
  ----------------
  $\forall\, tos : \mathrm{ran}(first\ InTempList?) \bullet$
      $\neg\ HasIntos(first\ tos)$

  We specify this condition in a separate schema to allow the reuse of DoElseStmt and
  InvalidTempList for the Meet operation (see below).
```

If no matching tuple has been found, the templates together with the requesting process will be added to the bag of pending templates, provided that no `else` statements are specified:

$$\begin{array}{l}
\underline{\mathit{FetchNoMatch}} \\
\Delta(\mathit{TSs}, \mathit{ActiveProcs}) \\
\mathit{InTempList?} : \mathit{TempList} \\
\mathit{InProc?} : \mathit{Process} \\
\mathit{Else?} : \mathit{OptStmt} \\
\hline
\mathit{second}\ \mathit{InTempList?} \in \mathit{IDsOF}\ \mathit{TSs} \\
\neg\,(\exists\, ts : \mathit{TSs} \mid (ts.\mathit{Id} = \mathit{second}\ \mathit{InTempList?}) \bullet \\
\quad \exists\, tos : \mathrm{ran}(\mathit{first}\ \mathit{InTempList?});\ tup : \mathrm{dom}\, ts.\mathit{Tuples} \bullet tup\ \mathsf{Matches}\ (\mathit{first}\ tos)) \\
\mathit{Else?} = \mathit{NoStmt} \\
\mathit{TSs}' = \{\, ts : \mathit{TSs};\ ts' : \mathit{TupleSpace} \mid ts'.\mathit{Id} = ts.\mathit{Id} \wedge \\
\qquad ts'.\mathit{Limit} = ts.\mathit{Limit} \wedge \\
\qquad ts'.\mathit{Tuples} = ts.\mathit{Tuples} \wedge \\
\qquad ts'.\mathit{PendFull} = ts.\mathit{PendFull} \wedge \\
\qquad ts'.\mathit{PendTemp} = \mathbf{if}\ ts'.\mathit{Id} = \mathit{second}\ \mathit{InTempList?} \\
\qquad\qquad\qquad \mathbf{then}\ ts.\mathit{PendTemp}\ \uplus \\
\qquad\qquad\qquad\qquad \mathit{MakePends}(\mathit{InTempList?}, \mathit{InProc?}, \mathit{FetchOp}) \\
\qquad\qquad\qquad \mathbf{else}\ ts.\mathit{PendTemp} \\
\qquad\quad \bullet\ ts' \,\} \\
\mathit{ActiveProcs}' = \mathit{ActiveProcs} \setminus \{\mathit{InProc?}\}
\end{array}$$

$$\begin{array}{l}
\mathit{Fetch}\ \widehat{=}\ (\mathit{FetchMatch} \vee \mathit{FetchNoMatch} \vee \mathit{DoElseStmt} \vee \mathit{InvalidTempList}) \wedge \\
\qquad \mathit{DisallowIntos}
\end{array}$$

The set *Matchings* within *FetchMatch* contains all the matching tuples. They are collected within this set together with the corresponding template and the optional statement. One such collection (a Z-tuple) is then selected, for which the following actions are necessary:

(assign the tuple fields to the l-values in the corresponding formals (optional)) $\wedge$
(execute the associated statements (optional)) $\wedge$
(remove the matching tuple from the tuple space) $\wedge$
(if the tuple space was full and there are processes pending on this full tuple space:
  (remove one of these pending processes from *PendFull*) $\wedge$
  (add the corresponding tuple to the bag *Tuples*) $\wedge$
  (reactivate the pending process))

As we can see, fetching a tuple may reactivate a process that is blocked with a `deposit` operation on a full tuple space.

8.7.4 Meeting Tuples

This section defines the *Meet* operation. The informal specification is given in Sect. 5.4.3. Meeting tuples is similar to fetching tuples except that the matching tuple is not removed from the tuple space and that it may be changed via `into` expressions while meeting it:

Meeting

$\restriction(Meet)$

$FetchMeet$

MeetMatch

$$\begin{array}{l}
\Delta(TSs, ActiveProcs)\\
InTempList? : TempList\\
InProc? : Process
\end{array}$$

$$\begin{array}{l}
second\ InTempList? \in IDsOF\ TSs\\
\exists\, ts : TSs \mid (ts.Id = second\ InTempList?) \bullet\\
\quad \exists\, tos : \mathrm{ran}(first\ InTempList?);\ tup : \mathrm{dom}\, ts.Tuples \bullet tup\ \mathsf{Matches}\ (first\ tos)\\
\exists\, myts : TSs \mid (myts.Id = second\ InTempList?) \bullet\\
\mathbf{let}\ Matchings == \{\, TEMP : Template;\ TUP : APTuple;\ OS : OptStmt \mid\\
\qquad\qquad\qquad (\exists\, tos : \mathrm{ran}(first\ InTempList?);\ tup : \mathrm{dom}\, myts.Tuples \bullet\\
\qquad\qquad\qquad\quad tup\ \mathsf{Matches}\ (first\ tos) \land\\
\qquad\qquad\qquad\quad TEMP = first\ tos \land TUP = tup \land OS = second\ tos) \bullet\\
\qquad\qquad\qquad (TEMP, TUP, OS)\,\} \bullet\\
(\exists_1\, SelMatch : Matchings \bullet\\
\quad (GetTemp\ SelMatch)\ \mathsf{FormalAssign}\ (GetTup\ SelMatch) \land\\
\quad OptExecute\ (GetOS\ SelMatch) \land\\
\quad (\neg\ HasIntos(GetTemp\ SelMatch) \Rightarrow\\
\qquad (TSs' = \{\, ts : TSs;\ ts' : TupleSpace \mid ts.Id = ts'.Id \land\\
\qquad\qquad\qquad ts'.Limit = ts.Limit \land\\
\qquad\qquad\qquad ts'.Tuples = ts.Tuples \land\\
\qquad\qquad\qquad ts'.PendTemp = ts.PendTemp \land\\
\qquad\qquad\qquad ts'.PendFull = ts.PendFull \bullet ts'\,\} \land\\
\qquad ActiveProcs' = ActiveProcs)) \land\\
\quad (HasIntos(GetTemp\ SelMatch) \Rightarrow\\
\quad (\exists_1\, newTSs : \mathbb{F}\ TupleSpace \bullet\\
\qquad newTSs = \{\, ts : TSs;\ newts : TupleSpace \mid newts.Id = ts.Id \land\\
\qquad\qquad\qquad newts.Limit = ts.Limit \land\\
\qquad\qquad\qquad newts.Tuples = \mathbf{if}\ newts.Id = myts.Id\\
\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ ts.Tuples \uplus [\![GetTup\ SelMatch]\!]\\
\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ ts.Tuples \land\\
\qquad\qquad\qquad newts.PendTemp = ts.PendTemp \land\\
\qquad\qquad\qquad newts.PendFull = ts.PendFull \bullet newts\,\} \land\\
\qquad (\mathbf{let}\ ChangedTuple == (GetTemp\ SelMatch)\ \mathsf{EvalIntos}\ (GetTup\ SelMatch) \bullet\\
\qquad (TSs', ActiveProcs') =\\
\qquad\qquad (newTSs, ActiveProcs)\ \mathsf{AddTuple}\ (ChangedTuple, myts.Id)))))
\end{array}$$

If there exists a template in the given *InTempList?*, which Matches a tuple in the associated tuple space, then the following actions are necessary:

(assign the tuple fields to the *l*-values in the corresponding formals (optional)) $\land$
(execute the associated statements (optional)) $\land$
(if there are `intos`:
  (remove the matching tuple from the tuple space) $\land$
  (add the changed tuple via AddTuple))

If no `intos` are specified, our program state does not change. If `intos` are specified, the matching tuple has to be removed and the changed one has to be added. The addition of the changed tuple might satisfy other pending processes with templates matching the changed tuple.

If no matching tuple has been found, the templates together with the requesting process will be added to the bag of pending templates, provided that no `else` statements are specified:

MeetNoMatch

$$
\begin{array}{l}
\Delta(TSs, ActiveProcs)\\
InTempList? : TempList\\
InProc? : Process\\
Else? : OptStmt\\
\hline
second\ InTempList? \in IDsOF\ TSs\\
\neg\ (\exists\ ts : TSs \mid (ts.Id = second\ InTempList?) \bullet\\
\quad \exists\ tos : \mathrm{ran}(first\ InTempList?);\ tup : \mathrm{dom}\, ts.Tuples \bullet tup\ \mathsf{Matches}\ (first\ tos))\\
Else? = NoStmt\\
TSs' = \{\ ts : TSs;\ ts' : TupleSpace \mid ts'.Id = ts.Id \land\\
\qquad ts'.Limit = ts.Limit \land\\
\qquad ts'.Tuples = ts.Tuples \land\\
\qquad ts'.PendFull = ts.PendFull \land\\
\qquad ts'.PendTemp = \mathbf{if}\ ts'.Id = second\ InTempList?\\
\qquad\qquad\qquad \mathbf{then}\ ts.PendTemp \uplus\\
\qquad\qquad\qquad\qquad MakePends(InTempList?, InProc?, MeetOp)\\
\qquad\qquad\qquad \mathbf{else}\ ts.PendTemp\\
\qquad\quad \bullet ts'\ \}\\
ActiveProcs' = ActiveProcs \setminus \{InProc?\}
\end{array}
$$

MeetNoMatch is similar to *FetchNoMatch*, except that *MeetOp* and *FetchOp* are exchanged.

$Meet \mathrel{\widehat=} MeetMatch \lor MeetNoMatch \lor DoElseStmt \lor InvalidTempList$

8.7.5 Fairness of the Tuple-Space Operations

The tuple-space operations with the fairness constraints are now defined as follows:

TupleSpaceOperations

$$
\begin{array}{l}
\upharpoonright (Deposit, Fetch, Meet)\\
Depositing\ [\mathbf{remove}\ DepositOK, TSisFull, FullTSException, FullTSBlock,\\
\qquad\qquad DepositInvalid]\\
Fetching\ [\mathbf{remove}\ FetchMatch, FetchNoMatch, DoElseStmt, InvalidTempList,\\
\qquad\qquad DisallowIntos]\\
Meeting\ [\mathbf{remove}\ MeetMatch, MeetNoMatch, DoElseStmt, InvalidTempList]
\end{array}
$$

Weakly fair selection of processes that are pending for templates:

$$
\begin{array}{l}
\forall\ ts : TSs;\ pp : Pending \mid pp \sqsubseteq ts.PendTemp \bullet\\
\quad \Box\Diamond(\exists\ ots : TSs \mid ots.Id = ts.Id \bullet (\exists\ tup : \mathrm{dom}\, ots.Tuples \bullet tup\ \mathsf{Matches}\ pp.temp))\\
\qquad \Rightarrow \Diamond(\exists\ ots : TSs \mid ots.Id = ts.Id \bullet \neg\ (pp \sqsubseteq ots.PendTemp))
\end{array}
$$

Weakly fair selection of processes that are pending for full tuple spaces:

$$
\begin{array}{l}
\forall\ ts : TSs;\ b : Process \times APTuple \mid b \in ts.PendFull \bullet\\
\quad \Box\Diamond(\exists\ ots : TSs \mid ots.Id = ts.Id \bullet IntValueOf\ ots.Limit > BagSum\ ots.Tuples)\\
\qquad \Rightarrow \Diamond(\exists\ ots : TSs \mid ots.Id = ts.Id \bullet b \notin ots.PendFull)
\end{array}
$$

We use the name *ots* as an acronym for *other tuple spaces*.

When objects of this class are instantiated, only the three tuple-space operations are available to such instances of *TupleSpaceOperations*. Note that children of this class still have access to the state variables of *ProgramState*, but not to the removed auxiliary operations.

The reader is referred to Sect. 5.4.4 for an informal discussion of weakly fair selection in ProSet. An informal outline of the history invariant for selection of processes that are pending for templates follows:

(for all tuple spaces which contain a pending `meet` or `fetch`:
  (there exists always eventually a matching tuple)
    $\Rightarrow$ (this pending `meet` or `fetch` will be selected eventually))

Weakly fair selection of processes that are pending for full tuple spaces is very similar. Note that weakly fair selection of processes that are pending for templates does not apply to non-blocking `fetch` or `meet` operations, because the corresponding requesting processes do not occur within the *PendFetch* and *PendMeet* components of tuple spaces. In Chap. 9 we will design an implementation for our specification of weakly fair selection.

8.8 Program Execution

An execution of a program is an infinite sequence of program states, each one related to its immediate successor by one of the operation schemas defined above:

ProgramExecution

$\upharpoonright(\textit{Init}, \textit{ProgramTermination}, \textit{ProcessCreation}, \textit{ProcessTermination}, \textit{CreateTS}, \textit{ExistsTS}, \textit{ClearTS}, \textit{RemoveTS}, \textit{Deposit}, \textit{Fetch}, \textit{Meet}, \textit{NoOp})$

ProcessCreationTermination
TupleSpaceLibrary
TupleSpaceOperations

Internal computations of processes do not change the program state of our specification. We can model them via the operation *NoOp*:

NoOp

$\Box(\mathbf{op} = \textit{ProgramTermination} \Rightarrow \Box(\mathbf{op} = \textit{NoOp}))$

The entire class hierarchy from *ProgramState* to *ProgramExecution* is displayed in Fig. 8.3.

For any given program, there is a set of valid execution histories. The first state change in the sequence represents the initialization (*Init*). It is not required that programs terminate. However, they may terminate: after *ProgramTermination* only *NoOps* are allowed. We model terminating programs via infinite sequences to keep the specification simple.

We present an operational semantics which defines the possible state changes of our program state through the operations on this state. We describe concurrency as usual when operational semantics are used:

> "As usual when parallelism is specified by an operational semantics, concurrency is described by an arbitrary interleaving of a set of atomic transactions performed by the acting processes [Milner, 1989].
>
> The interleaving approximation does not provide true concurrency, but it is sufficient as tuples are indivisible units which are manipulated by atomic actions. [...] The possibility of blocking operations are modeled by the *match* relation: [...]" [Ciancarini *et al.*, 1992, page 7]

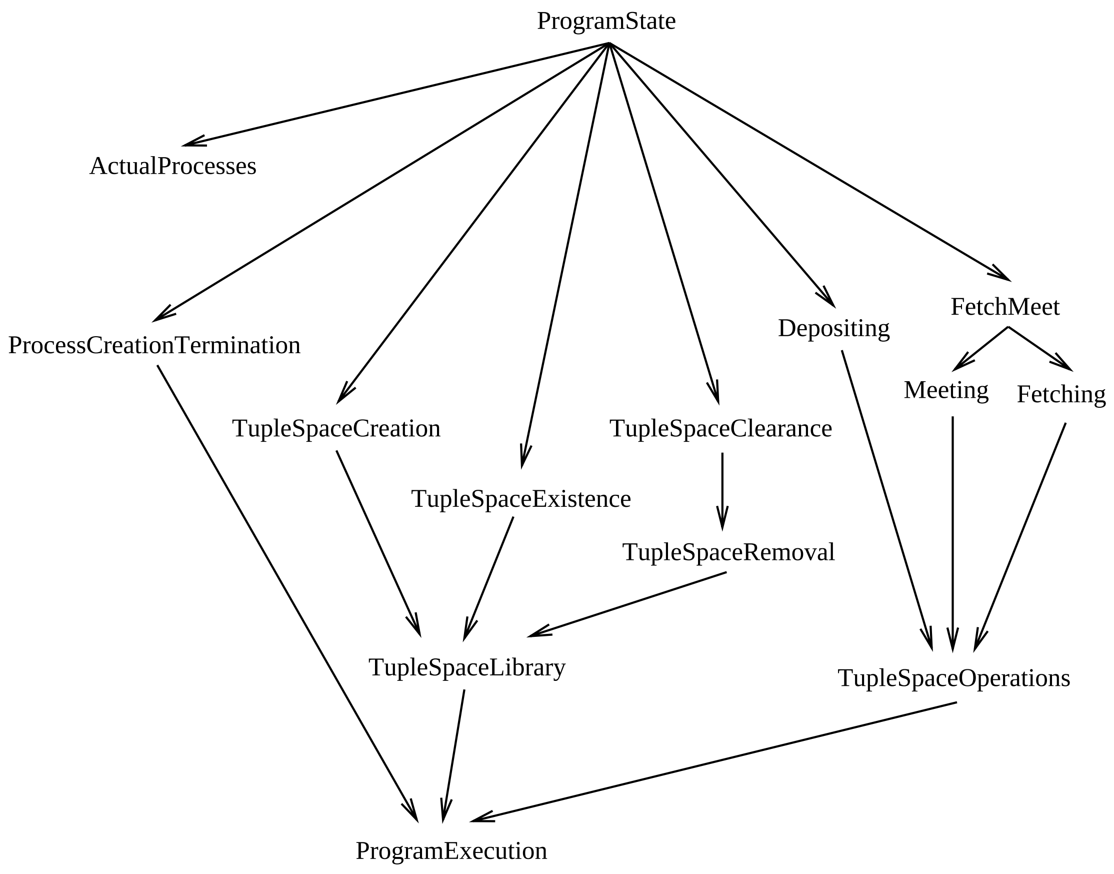


Figure 8.3: The entire class hierarchy for our formal specification.

Note that the notions of distribution and asynchrony are not captured by such an operational semantics. The goal of the present work is not to specify as much parallelism as possible, but rather to provide a precise specification of the semantics of generative communication in ProSet.

8.9 Summary

We presented the formal specification of generative communication in ProSet by means of the formal specification language Object-Z. Concurrency is described by arbitrary interleavings of the atomic actions of the participating processes. However, nothing in the semantics given here prevents causal independent actions to occur in parallel. The concurrency of programs is modeled by the nondeterministic interleaving of atomic actions, i.e., by an asynchronous model. Atomic transitions happen one after another in a non-fixed arbitrary order.

We specified formally the conditions under which exceptions have to be raised. The actions to be taken for handling such situations were only sketched. This is due to the fact that we only present a specification of generative communication in ProSet, and not a specification of the entire language. A rigorous formal specification of exception handling is in general not a light-weight exercise.

The concept for process creation in ProSet is adapted from Multilisp's futures. The resolving and touching of futures is only specified for processes within active tuples in tuple space, and not for processes spawned outside of it. This limitation is also due to our not specifying the entire language.

The specification of the formal semantics of generative communication in ProSet led us to the recognition of several omissions and imperfections in our previous informal specification, which has been presented in its first version in [Hasselbring, 1991a] and in a revised version in [Hasselbring, 1991b]. Nevertheless, the main advantage of using a formal specification lies in subsequent development steps for the implementation. Formal specifications may be needed as an intermediate step between requirement analysis and design. The formal specification defines the duties for the implementor in a precise and unambiguous way. The implementation design of Chap. 9 is the first step for implementing ProSet-Linda.

Part III

Implementation

Chapter 9

Refining the Formal Specification

Once the formal semantics of ProSet-Linda has been given, the next step is to consider the properties of an implementation design. Obtaining a design for an implementation from a specification can be done via top-down, step-wise data and operation refinement. Data refinement is the process of transforming one data type into another one: an abstractly given data type is transformed into a more concrete one. This is the way we approach the implementation. Refinement means adding more and more implementation details (for example, choosing a particular algorithm or data representation) until an executable implementation is achieved.

In this chapter we will refine our formal specification to obtain the implementation design for our formal specification of Chap. 8. We start in Sect. 9.1 with a general discussion of the relationship between the specification of the abstract semantics and the specification of the more concrete design. Data and operation refinement for our formal specification is then presented in Sects. 9.2 and 9.3, respectively. The correctness of the refinement is discussed in Sect. 9.4.

9.1 Semantics versus Implementation Design

We present the formal semantics of ProSet-Linda in Chap. 8 on a very high level without considering a possible implementation. It is good practice to write another specification which uses less abstract data types and more *algorithmic* constructs to approach the implementation [Diller, 1990]. This second specification can then be thought of as being intermediate between the original specification and an executable implementation. In the remainder of this chapter we will refer to the abstract specification as the *semantics* and to the more concrete specification as the *design*.

What we require is that any program which is a correct implementation of the design is also a correct implementation of the semantics. When this requirement is satisfied, then the operations in the design *model* those in the semantics [Diller, 1990]. For data refinement an abstraction relation between the abstract state space of the semantics and the concrete state space of the design has to be given (see Sect. 9.2). Operation refinement leads to algorithm development. We refer to [Diller, 1990, Chapter 13], [Woodcock, 1991], and [Spivey, 1992b, Sections 1.5 and 5.6] for introductory examples for refinement in Z.

It is quite usual for one abstract state in the semantics to be represented by many concrete states in the design. As an example, finite sets can be represented by sequences in which the order of elements does not matter; in such a representation, a set of size n can be represented by any of $n!$ different sequences with the elements in different orders (see [Spivey, 1992b, Sect. 5.6] for an example).

Especially, nondeterminism may be restricted in the design. However, the design must satisfy two requirements to be considered correct:

liveness If a set of processes is able to act in the context of the semantics, then at least one of them must also be able to act in the context of the design. This implies that whenever a process in the context of the semantics is guaranteed to terminate, the process also terminates in the context of the design.

deadlock If no process can act in the context of the semantics, then no process may be able to act in the context of the design. The semantics and the design must have the same deadlock properties.

9.2 Data Refinement

The weak fairness property for pending processes in the semantics as formally specified in Sect. 8.7.5 can be implemented by using FIFO queues for pending processes instead of bags and sets as in the definition of *TupleSpace* on page 93. In a FIFO (first-in first-out) queue always the possible candidate who waits longest is selected first. The design for tuple spaces uses sequences, which are used as FIFO queues, for pending processes:

$$
\begin{array}{|l}
\hline
\textit{TupleSpaceD} \\
\textit{IdD} : \textit{Value} \\
\textit{LimitD} : \textit{Value} \\
\textit{TuplesD} : \mathrm{bag}\ \textit{APTuple} \\
\textit{PendTempD} : \mathrm{seq}\ \textit{Pending} \\
\textit{PendFullD} : \mathrm{iseq}(\textit{Process} \times \textit{APTuple}) \\
\hline
(\textit{Type IdD} = \textit{atom}) \wedge (\textit{IdD} \notin \mathrm{dom}\ \textit{ValuesOfType}) \\
(\textit{Type LimitD} = \textit{integer}) \vee (\textit{Type LimitD} = \textit{om}) \\
\mathsf{disjoint}\ \langle \{\, pt : \mathrm{ran}\ \textit{PendTempD} \bullet pt.proc \,\}, \mathrm{dom}(\mathrm{ran}\ \textit{PendFullD}) \rangle \\
\hline
\end{array}
$$

iseq X is the set of injective finite sequences over X: these are precisely the finite sequences over X which contain no repetitions.

We implement the bag *PendTemp* through the sequence *PendTempD* (repetitions are allowed in sequences) and the set *PendFull* through the injective sequence *PendFullD* (repetitions are not allowed in injective sequences). We will use these sequences as FIFO queues in the concrete operations of Sect. 9.3. Items will be appended to the tails and removed from the heads of these queues.

The definitions for the design are distinguished from those of the semantics through an appended D. We document the correspondence between semantics and design with a schema *TSAbstraction* that defines the *abstraction relation* between the abstract state space *TupleSpace* and the concrete state space *TupleSpaceD*:

$$
\begin{array}{|l}
\hline
\textit{TSAbstraction} \\
\textit{TupleSpace} \\
\textit{TupleSpaceD} \\
\hline
\textit{Id} = \textit{IdD} \\
\textit{Limit} = \textit{LimitD} \\
\textit{Tuples} = \textit{TuplesD} \\
\textit{PendTemp} = \textit{items PendTempD} \\
\textit{PendFull} = \mathrm{ran}\ \textit{PendFullD} \\
\hline
\end{array}
$$

If s is a sequence, *items* s is the bag in which each element x appears exactly as often as x appears in s.

The abstraction relation is also called *retrieve relation schema* because it allows us to retrieve the abstraction from the implementation details [Litteck and Wallis, 1992]. The relation between abstract tuple spaces and concrete ones can now be formulated as follows [Spivey, 1992b]:

$$\forall\ TupleSpaceD \bullet \exists_1\ TupleSpace \bullet TSAbstraction$$

Therefore, there exists exactly one abstract state for each concrete state, and there may exist several concrete states for each abstract state: the retrieve relation is a total function from concrete to abstract states.

The concrete program state is defined as follows:

ProgramStateD

$TSsD : \mathbb{F}\ TupleSpaceD$
$ActiveProcsD : \mathbb{F}\ Process$

$\forall\ ts1, ts2 : TSsD \bullet$
$\quad ts1 \neq ts2 \Rightarrow ts1.IdD \neq ts2.IdD$
$\forall\ ts1, ts2 : TSsD \bullet$
$\quad \forall\ tup1, tup2 : \mathrm{dom}(ts1.TuplesD \uplus ts2.TuplesD);\ i1, i2 : \mathbb{N} \mid$
$\qquad\qquad tup1(i1) \in \mathrm{ran}\ TupleProcess \wedge tup2(i2) \in \mathrm{ran}\ TupleProcess \bullet$
$\qquad (i1 \in \mathrm{dom}\ tup1 \wedge i2 \in \mathrm{dom}\ tup2 \wedge tup1(i1) = tup2(i2)) \Rightarrow$
$\qquad\quad tup1 = tup2 \wedge i1 = i2 \wedge$
$\qquad\quad 1 = (ts1.TuplesD \uplus ts2.TuplesD) \sharp\ tup1$

Only names were changed compared to the definition of *ProgramState* on page 95. Again, we document the correspondence between semantics and design with a schema *ProgramAbstraction* that defines the *abstraction relation* between the abstract program state *ProgramState* and the concrete state space *ProgramStateD*:

ProgramAbstraction

$ProgramState$
$ProgramStateD$

$\forall\ tsd : TSsD \bullet \exists_1\ ts : TSs \bullet$
$\quad ts.Id = tsd.IdD \wedge$
$\quad ts.Limit = tsd.LimitD \wedge$
$\quad ts.Tuples = tsd.TuplesD \wedge$
$\quad ts.PendTemp = items\ tsd.PendTempD \wedge$
$\quad ts.PendFull = \mathrm{ran}\ tsd.PendFullD$
$ActiveProcs = ActiveProcsD$

The relation between abstract program state and concrete program state can now be formulated as follows:

$$\forall\ ProgramStateD \bullet \exists_1\ ProgramState \bullet ProgramAbstraction$$

This is the retrieve relation for program states. It is a total function from concrete to abstract program states.

9.3 Operation Refinement

Having explained what the concrete state space is, and how concrete and abstract state spaces are related, we can begin to give designs for the operations of the semantics.

As noted earlier, we use FIFO queues in our design for pending processes and model them in Z through sequences. The tuple-space manager has to handle these queues in such a way that the fairness properties of Sect. 8.7.5 are satisfied.

There are four kinds of operations which are involved in handling these queues:

1. `deposit` operations which can add a tuple (the requested tuple space is not full and no exception prevents the addition):

   Pending processes with matching templates shall be removed from the *PendFullD* queue (provided that there are such processes pending):

   - All processes with `meet` operations and *no* `into`s are selected.
   - The first process with a `fetch` operation or a `meet` operation with `into`s is selected.

   See the definition of AddTupleD for *DepositOKD* below.

2. `deposit` operations which cannot add a tuple because the requested tuple space is full and `blockiffull` has been specified:

   - The requesting process is appended to the end of the corresponding *PendFullD* queue.

   See the definition of *FullTSBlockD* below.

3. `fetch` operations for which matching tuples are available in a full tuple space:

   - The first process which is blocked on this full tuple space with a `deposit` operation is selected and removed from the queue (provided that there is such a process pending).

   See the definition of *FetchMatchD* below.

4. Blocking `fetch` or `meet` operations for which no matching tuple is available (no `else` statements specified):

   - The requesting process is appended to the end of the corresponding *PendTempD* queue.

   See the definitions of *FetchNoMatchD* and *MeetNoMatchD* below.

Fig. 9.1 illustrates the management of the queues for processes which are blocked with blocking `fetch` or `meet` operations, and Fig. 9.2 illustrates the management of the queues for processes which are blocked on full tuple spaces. We will present in the subsequent subsections only the operations *DepositOKD*, *FullTSBlockD*, *FetchMatchD*, and *FetchNoMatchD* to save space. No significant changes are necessary in the remaining operations which belong to the tuple-space operations, in the operations for program/process initialization and termination, in the operations for the library functions to handle multiple tuple spaces, and for program execution: only names are changed.

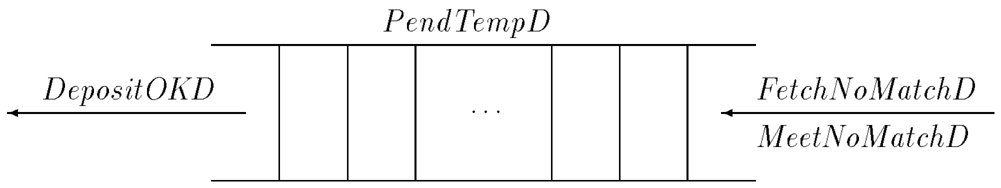


Figure 9.1: The queue for blocking `fetch` and `meet` operations.
Processes which will be blocked because no matching tuple is available, are appended by *FetchNoMatchD* or *MeetNoMatchD* to the tail of the *PendTempD* queue. *DepositOKD* takes from *PendTempD* the first template for a `fetch` operation or a `meet` operation with `into`s which matches the tuple to be deposited. This is done via AddTupleD. Processes which are blocked for `meet` operations without `into`s are selected without consideration of their position in the queue. We select all processes for such `meet` operations.

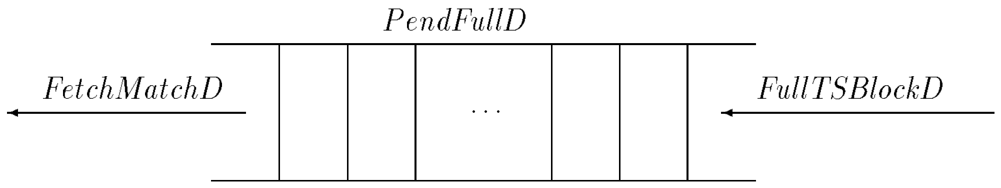


Figure 9.2: The queue for processes which are blocked on full tuple spaces.
Processes which will be blocked because a tuple space is full, are appended by *FullTSBlockD* to the tail of the *PendFullD* queue. After fetching a tuple *FetchMatchD* takes from the head of *PendFullD* the first blocked process which wants to deposit a tuple.

9.3.1 Some Preliminary Definitions

We will need the auxiliary function *IDsOFD* for checking tuple-space identities, which is the concrete counterpart of *IDsOF* (page 94):

$$
\begin{array}{|l}
IDsOFD : \mathbb{F}\, TupleSpaceD \longrightarrow \mathbb{F}\, Value \\
\hline
\forall\, tss : \mathbb{F}\, TupleSpaceD \bullet \\
\qquad IDsOFD\ tss = \{\, id : Value \mid (\exists\, ts : tss \bullet ts.IdD = id) \,\}
\end{array}
$$

Again, we will need an auxiliary function for schema anti-restriction for sequences of pending processes:

$$
\begin{array}{|l}
\_\boxed{|||}\_ : \mathbb{F}\, Process \times \operatorname{seq} Pending \longrightarrow \operatorname{seq} Pending \\
\hline
\forall\, procs : \mathbb{F}\, Process;\ pends : \operatorname{seq} Pending \bullet \\
\qquad procs \boxed{|||} pends = \\
\qquad\qquad squash\{\, pe : Pending;\ i : \mathbb{N} \mid pe = pends(i) \wedge pe.proc \notin procs \bullet i \mapsto pe \,\}
\end{array}
$$

$\boxed{Z}$ The sequence compaction *squash* takes a finite function defined on the strictly positive integers and compacts it into a sequence by removing gaps in the domain of the given finite function: $squash\{\, 1 \mapsto 123, 3 \mapsto 456 \,\} = \{\, 1 \mapsto 123, 2 \mapsto 456 \,\}$ $\boxed{Z}$

The corresponding function $\boxed{|||}$ for bags is defined on page 94.

The auxiliary function *MakePendsD* creates from a *TempList*, a process, and the operation type a corresponding sequence of pending processes:

$$
\begin{array}{|l}
MakePendsD : TempList \times Process \times OpType \longrightarrow \operatorname{seq} Pending \\
\hline
\forall\, tl : TempList;\ tos : Template \times OptStmt;\ tosl : \operatorname{seq}(Template \times OptStmt); \\
\quad id : Value;\ pr : Process;\ ot : OpType \bullet \\
\qquad MakePendsD((\langle\rangle, id), pr, ot) = \langle\rangle \wedge \\
\qquad MakePendsD((tosl \frown \langle tos \rangle, id), pr, ot) = MakePendsD((tosl, id), pr, ot) \frown \\
\qquad\qquad \{\, pe : Pending \mid pe.proc = pr \wedge pe.temp = first\ tos \wedge pe.os = second\ tos \wedge \\
\qquad\qquad\qquad pe.type = \mathbf{if}\ HasIntos(first\ tos) \\
\qquad\qquad\qquad\qquad \mathbf{then}\ MeetIntoOp \\
\qquad\qquad\qquad\qquad \mathbf{else}\ ot \qquad \bullet 1 \mapsto pe \,\}
\end{array}
$$

The function *MakePends* on page 103 creates the corresponding bag. Note that the order in which multiple templates of *one* operation are appended is not important. We retain the order in the given *TempList*.

We define the function AddTupleD for adding a single tuple to a specified tuple space of the concrete state (see also AddTuple for the abstract state on page 104):

$$
\begin{array}{l}
\_\,\mathsf{AddTupleD}\,\_ : (\mathbb{F}\ TupleSpaceD \times \mathbb{F}\ Process) \times (APTuple \times Value) \\
\qquad\qquad\qquad\qquad\qquad\qquad \longrightarrow (\mathbb{F}\ TupleSpaceD \times \mathbb{F}\ Process) \\
\hline
\forall\, tss, tss' : \mathbb{F}\ TupleSpaceD;\ AP, AP' : \mathbb{F}\ Process;\ tup : APTuple;\ id : Value \bullet \\
\quad (tss', AP') = (tss, AP)\ \mathsf{AddTupleD}\ (tup, id) \Leftrightarrow \\
\qquad (\exists_1\, myts : tss \mid myts.IdD = id \bullet \\
\qquad (\mathbf{let}\ MatchMeets == \{\, pt : \operatorname{ran} myts.PendTempD \mid tup\ \mathsf{Matches}\ pt.temp \wedge \\
\qquad\qquad\qquad\qquad pt.type = MeetOp \,\}; \\
\qquad\quad NewProcs == \{\, p : Process \mid (\exists\, tupc : \operatorname{ran} tup \bullet tupc = TupleProcess\ p) \,\} \bullet \\
\qquad (\forall\, pm : \mathbb{P}\ MatchMeets \mid (\forall\, p1, p2 : pm \bullet p1.proc = p2.proc) \bullet \exists_1\, pend : pm \bullet \\
\qquad\quad pend.temp\ \mathsf{FormalAssign}\ tup \wedge \\
\qquad\quad OptExecute\ pend.os) \wedge \\
\qquad (\mathbf{let}\ MeetProcs == \{\, mm : MatchMeets \bullet mm.proc \,\} \bullet \\
\qquad (\mathbf{let}\ MatchIntosFetchs == MeetProcs \mathbin{▥} myts.PendTempD \bullet \\
\qquad ((MatchIntosFetchs = \{\}) \Rightarrow \\
\qquad\quad (tss' = \{\, ts : tss;\ ts' : TupleSpaceD \mid ts'.IdD = ts.IdD \wedge \\
\qquad\qquad\qquad ts'.LimitD = ts.LimitD \wedge \\
\qquad\qquad\qquad ts'.TuplesD = \mathbf{if}\ ts'.IdD = id \\
\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ ts.TuplesD \uplus \llbracket tup \rrbracket \\
\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ ts.TuplesD \wedge \\
\qquad\qquad\qquad ts'.PendTempD = MeetProcs \mathbin{▥} ts.PendTempD \wedge \\
\qquad\qquad\qquad ts'.PendFullD = ts.PendFullD \bullet ts' \,\} \wedge \\
\qquad\quad AP' = AP \cup NewProcs \cup MeetProcs)) \wedge \\
\qquad ((MatchIntosFetchs \neq \{\}) \Rightarrow \\
\qquad (\mathbf{let}\ mif == head\ MatchIntosFetchs \bullet \\
\qquad\quad mif.temp\ \mathsf{FormalAssign}\ tup \wedge \\
\qquad\quad OptExecute\ mif.os \wedge \\
\qquad\quad (mif.type = FetchOp \Rightarrow \\
\qquad\qquad (tss' = \{\, ts : tss;\ ts' : TupleSpaceD \mid ts'.IdD = ts.IdD \wedge \\
\qquad\qquad\qquad ts'.LimitD = ts.LimitD \wedge \\
\qquad\qquad\qquad ts'.TuplesD = ts.TuplesD \wedge \\
\qquad\qquad\qquad ts'.PendTempD = \\
\qquad\qquad\qquad\quad (MeetProcs \cup \{mif.proc\}) \mathbin{▥} ts.PendTempD \wedge \\
\qquad\qquad\qquad ts'.PendFullD = ts.PendFullD \bullet ts' \,\} \wedge \\
\qquad\qquad AP' = AP \cup MeetProcs \cup \{mif.proc\})) \wedge \\
\qquad\quad (mif.type = MeetIntoOp \Rightarrow \\
\qquad\quad (\exists_1\, newTSs : \mathbb{F}\ TupleSpaceD;\ newAP : \mathbb{F}\ Process \bullet \\
\qquad\qquad newTSs = \{\, ts : tss;\ newts : TupleSpaceD \mid newts.IdD = ts.IdD \wedge \\
\qquad\qquad\qquad newts.LimitD = ts.LimitD \wedge \\
\qquad\qquad\qquad newts.TuplesD = ts.TuplesD \wedge \\
\qquad\qquad\qquad newts.PendTempD = \\
\qquad\qquad\qquad\quad (MeetProcs \cup \{mif.proc\}) \mathbin{▥} ts.PendTempD \wedge \\
\qquad\qquad\qquad newts.PendFullD = ts.PendFullD \bullet newts \,\} \wedge \\
\qquad\qquad newAP = AP \cup MeetProcs \cup \{mif.proc\} \wedge \\
\qquad\qquad (tss', AP') = \\
\qquad\qquad\quad (newTSs, newAP)\ \mathsf{AddTupleD}\ (mif.temp\ \mathsf{EvalIntos}\ tup, id)))))))))
\end{array}
$$

$\boxed{Z}$ For a non-empty sequence s, $head\ s$ is the first element of s. $\boxed{Z}$

The informal description of AddTuple can essentially be reused for AddTupleD. *MatchIntos* and *MatchFetchs* are replaced by *MatchIntosFetchs*. The main difference lies in the way *mif* is selected from the matching `meet` templates with `into`s and the matching `fetch` templates: *mif* is selected from the head of *MatchIntosFetchs*. This way *PendTempD* is used as a FIFO queue as illustrated in Fig. 9.1, because the order of items in *PendTempD* is sustained in *MatchIntosFetchs*. *MatchIntosFetchs* is a subsequence of *PendTempD*.

Note that processes which are blocked for `meet` operations without `into`s are selected without consideration of their position in the queue. We select all processes for such `meet` operations, but each process only once.

9.3.2 Depositing Tuples

We only present the components *DepositOKD* and *FullTSBlockD* of *DepositingD* (see page 106 for *Depositing*):

DepositingD

$ProgramStateD$

DepositOKD

$\Delta(TSsD, ActiveProcsD)$
$ToDeposit? : APTuple \times Value$

$second\ ToDeposit? \in IDsOFD\ TSsD$
$(TSsD', ActiveProcsD') = (TSsD, ActiveProcsD)\ \mathsf{AddTupleD}\ ToDeposit?$

Processes to be blocked on a full tuple space are appended to the tail of the corresponding queue:

FullTSBlockD

$\Delta(TSsD, ActiveProcsD)$
$ToDeposit? : APTuple \times Value$
$InProc? : Process$
$Blocking? : BlockMode$

$second\ ToDeposit? \in IDsOFD\ TSsD$
$Blocking? = BlockIfFull$

$$\begin{aligned}TSsD' = \{\, & ts : TSsD;\ ts' : TupleSpaceD \mid ts'.IdD = ts.IdD \wedge \\ & ts'.LimitD = ts.LimitD \wedge \\ & ts'.TuplesD = ts.TuplesD \wedge \\ & ts'.PendTempD = ts.PendTempD \wedge \\ & ts'.PendFullD = \mathbf{if}\ ts'.IdD = second\ ToDeposit? \\ & \qquad \mathbf{then}\ ts.PendFullD \frown \langle InProc? \mapsto first\ ToDeposit? \rangle \\ & \qquad \mathbf{else}\ ts.PendFullD \bullet ts' \,\}\end{aligned}$$

$ActiveProcsD' = ActiveProcsD \setminus \{InProc?\}$

The definitions for *DepositInvalidD*, *TSisFullD*, *FullTSExceptionD*, and *DepositD* are not shown to save space.

The difference between *FullTSBlockD* and *FullTSBlock* is that the process, which will be blocked on a full tuple space, is appended to the tail of the *PendFullD* queue. This way *PendFullD* is used as a FIFO queue as illustrated in Fig. 9.2.

9.3.3 Fetching Tuples

We only present the components *FetchMatchD* and *FetchNoMatchD* of *FetchingD* (see page 108 for *Fetching*):

```
FetchingD
$ProgramStateD$

    FetchMatchD
    $\Delta(TSsD, ActiveProcsD)$
    $InTempList? : TempList$
    $InProc? : Process$
    ----------------
    $second\ InTempList? \in IDsOFD\ TSsD$
    $\exists\, ts : TSsD \mid (ts.IdD = second\ InTempList?) \bullet$
        $\exists\, tos : \mathrm{ran}(first\ InTempList?);\ tup : \mathrm{dom}\, ts.TuplesD \bullet tup\ \mathsf{Matches}\ (first\ tos)$
    $\exists\, myts : TSsD \mid (myts.IdD = second\ InTempList?) \bullet$
    $\mathbf{let}\ Matchings == \{\ TEMP : Template;\ TUP : APTuple;\ OS : OptStmt \mid$
                        $(\exists\, tos : \mathrm{ran}(first\ InTempList?);\ tup : \mathrm{dom}\, myts.TuplesD \bullet$
                            $tup\ \mathsf{Matches}\ (first\ tos) \land$
                            $TEMP = first\ tos \land TUP = tup \land OS = second\ tos) \bullet$
                        $(TEMP, TUP, OS)\ \} \bullet$
    $(\exists_1\, SelMatch : Matchings \bullet$
        $(GetTemp\ SelMatch)\ \mathsf{FormalAssign}\ (GetTup\ SelMatch) \land$
        $OptExecute\ (GetOS\ SelMatch) \land$
        $((myts.PendFullD = \langle\rangle) \Rightarrow$
            $(TSsD' = \{\ ts : TSsD;\ ts' : TupleSpaceD \mid ts.IdD = ts'.IdD \land$
                        $ts'.LimitD = ts.LimitD \land$
                        $ts'.TuplesD = \mathbf{if}\ ts'.IdD = myts.IdD$
                                    $\mathbf{then}\ ts.TuplesD \uplus [\![GetTup\ SelMatch]\!]$
                                    $\mathbf{else}\ ts.TuplesD \land$
                        $ts'.PendTempD = ts.PendTempD \land$
                        $ts'.PendFullD = ts.PendFullD \bullet ts'\ \} \land$
            $ActiveProcsD' = ActiveProcsD)) \land$
        $((myts.PendFullD \neq \langle\rangle) \Rightarrow$
        $(\mathbf{let}\ SelBlocked == head\ myts.PendFullD \bullet$
            $(TSsD' = \{\ ts : TSsD;\ ts' : TupleSpaceD \mid ts.IdD = ts'.IdD \land$
                        $ts'.LimitD = ts.LimitD \land$
                        $ts'.TuplesD = \mathbf{if}\ ts'.IdD = myts.IdD$
                                    $\mathbf{then}\ (ts.TuplesD \uplus [\![GetTup\ SelMatch]\!])$
                                            $\uplus [\![second\ SelBlocked]\!]$
                                    $\mathbf{else}\ ts.TuplesD \land$
                        $ts'.PendTempD = ts.PendTempD \land$
                        $ts'.PendFullD = \mathbf{if}\ ts'.IdD = myts.IdD$
                                    $\mathbf{then}\ tail\ ts.PendFullD$
                                    $\mathbf{else}\ ts.PendFullD \bullet ts'\ \} \land$
            $ActiveProcsD' = ActiveProcsD \cup \{first\ SelBlocked\}))))$
```

$\boxed{Z}$ For a non-empty sequence s, the sequence $tail\ s$ contains all the elements of s except for the first one (in the same order). $\boxed{Z}$

Note that it is not necessary to select the templates of *one* operation in a fair way, when a matching tuple is available. Also, the selection of matching tuples is unfair, but processes which are blocked on this full tuple space are taken from the head of the *PendFullD* queue (see the way in which *SelBlocked* is selected).

If no matching tuple has been found, the templates together with the requesting process will be appended to the tail of the queue of pending templates, provided that no `else` statements are specified:

FetchNoMatchD

$$\begin{array}{l}
\Delta(\mathit{TSsD}, \mathit{ActiveProcsD}) \\
\mathit{InTempList}? : \mathit{TempList} \\
\mathit{InProc}? : \mathit{Process} \\
\mathit{Else}? : \mathit{OptStmt} \\
\hline
\mathit{second}\ \mathit{InTempList}? \in \mathit{IDsOFD}\ \mathit{TSsD} \\
\neg\,(\exists\, ts : \mathit{TSsD} \mid (ts.\mathit{IdD} = \mathit{second}\ \mathit{InTempList}?) \bullet \\
\quad \exists\, tos : \mathrm{ran}(\mathit{first}\ \mathit{InTempList}?);\ tup : \mathrm{dom}\, ts.\mathit{TuplesD} \bullet tup\ \mathsf{Matches}\ (\mathit{first}\ tos)) \\
\mathit{Else}? = \mathit{NoStmt} \\
\mathit{TSsD}' = \{\, ts : \mathit{TSsD};\ ts' : \mathit{TupleSpaceD} \mid ts'.\mathit{IdD} = ts.\mathit{IdD} \wedge \\
\qquad ts'.\mathit{LimitD} = ts.\mathit{LimitD} \wedge \\
\qquad ts'.\mathit{TuplesD} = ts.\mathit{TuplesD} \wedge \\
\qquad ts'.\mathit{PendFullD} = ts.\mathit{PendFullD} \wedge \\
\qquad ts'.\mathit{PendTempD} = \mathbf{if}\ ts'.\mathit{IdD} = \mathit{second}\ \mathit{InTempList}? \\
\qquad\qquad\qquad\qquad \mathbf{then}\ ts.\mathit{PendTempD}\ ^\frown \\
\qquad\qquad\qquad\qquad\quad \mathit{MakePendsD}(\mathit{InTempList}?, \mathit{InProc}?, \mathit{FetchOp}) \\
\qquad\qquad\qquad\qquad \mathbf{else}\ ts.\mathit{PendTempD} \\
\qquad\quad \bullet\ ts'\,\} \\
\mathit{ActiveProcsD}' = \mathit{ActiveProcsD} \setminus \{\mathit{InProc}?\}
\end{array}$$

The definitions for *DoElseStmtD*, *InvalidTempListD*, and *FetchD* are not shown to save space.

The difference between *FetchMatchD* and *FetchMatch* is that the process, which may be selected from the *PendFullD* queue, is selected from the head of *PendFullD*. This way *PendFullD* is used as a FIFO queue as illustrated in Fig. 9.2.

The difference between *FetchNoMatchD* and *FetchNoMatch* is that the process, which will be blocked because no matching tuple is available, is appended to the tail of the respective *PendTempD* queue. This way *PendTempD* is used as a FIFO queue as illustrated in Fig. 9.1.

9.3.4 Meeting Tuples

MeetNoMatchD is similar to *FetchNoMatchD*, except that *MeetOp* and *FetchOp* are exchanged, and that in *MeetMatchD* only *D*s are appended to the names. Therefore, we do not present *MeetingD* to save space.

9.4 Correctness of the Refinement

We have to prove that the concrete operations are correct implementations of the abstract operations. What we require is that any program which is a correct implementation of the design is also a correct implementation of the semantics. When this requirement is satisfied we say that the concrete state of the design is a *reification* [Diller, 1990] of the abstract state of the semantics and that the operations in the design *model* those in the semantics.

Essentially, we implement bags (*PendTemp*) through sequences (*PendTempD*) and sets (*PendFull*) through injective sequences (*PendFullD*). One standard way of proving things about sequences is by using *sequence induction*, which proceeds according to the cardinality of the sequences under consideration. In [Diller, 1990, Chap. 13] it is shown how sets can be implemented through injective sequences, and how the correctness can be proven by using sequence induction. We do not repeat this proof here. Instead we prove the correctness of implementing bags through sequences (*PendTemp* through *PendTempD*) in Sect. 9.4.1. The satisfaction of the fairness properties is shown in Sect. 9.4.2.

9.4.1 Modeling Bags by Sequences

We have to show that *FetchNoMatchD* is a correct implementation of *FetchNoMatch*, that *MeetNoMatchD* is a correct implementation of *MeetNoMatch*, and that *DepositOKD* is a correct implementation of *DepositOK*.

The way we relate the abstract and concrete data types is by means of a *retrieve function*. The retrieve function *ret* maps the concrete type into the abstract one:

$$ret : \text{seq}\ Pending \longrightarrow \text{bag}\ Pending$$

In the case of bags being modeled as sequences the function *items*, which is used in the retrieve relation schema *TSAbstraction* in Sect. 9.2, acts as the retrieve function. The retrieve function has to map the concrete type to the abstract one because many sequences correspond to the same bag.

Processes are appended by *FetchNoMatchD* and *MeetNoMatchD* to the tail of the *PendTempD* queue when they will be blocked because no matching tuple is available. We have to prove the following in order to show that this is a correct modeling of *FetchNoMatch* and *MeetNoMatch*:

$$\begin{aligned}&\forall\, ProgramStateD;\ pends : \text{seq}_1\ Pending \bullet \forall\, ts : TSsD \bullet\\ &\qquad items(ts.PendTempD \frown pends) = items\ ts.PendTempD \uplus items\ pends\end{aligned}$$

The concrete operations use sequence concatenation ($\frown$) and the abstract operations use bag union ($\uplus$) to add new pending processes to the queues for blocking `fetch` and `meet` operations.

Proof: The following identity is always valid for two sequences s and t [Spivey, 1992b, page 127]:

$$items(s \frown t) = items\ s \uplus items\ t$$

This identity shows that *FetchNoMatchD* is a correct implementation of *FetchNoMatch* and that *MeetNoMatchD* is a correct implementation of *MeetNoMatch*. □

Processes are removed by *DepositOKD* from the *PendTempD* queue when the associated templates match the deposited tuple. *DepositOKD* takes via AddTupleD the first template for a `fetch` operation or a `meet` operation with `into`s which matches the tuple to be deposited. Processes which are blocked for `meet` operations without `into`s are selected without consideration of their position in the queue. We select all processes for such `meet` operations. We have to prove the following in order to show that this is a correct modeling of *DepositOK*:

$$\begin{aligned}&\forall\, ProgramStateD;\ procs : \mathbb{F}\ Process \bullet \forall\, ts : TSsD \bullet\\ &\qquad items(procs ▥ ts.PendTempD) = procs ◫ items\ ts.PendTempD\end{aligned}$$

DepositOKD uses the schema anti-restriction ▥ (defined on page 122) and *DepositOK* uses the schema anti-restriction ◫ (defined on page 94) to remove pending processes, whose templates match the tuple to be deposited, from the queues for blocking `fetch` and `meet` operations. We prove this by sequence induction:

Base case: We need to prove that the property is true for the empty set $\{\}$:

$$\begin{aligned}&\forall\, ProgramStateD \bullet \forall\, ts : TSsD \bullet\\ &\qquad items(\{\} ▥ ts.PendTempD) = \{\} ◫ items\ ts.PendTempD\end{aligned}$$

The base case is obviously satisfied because $\{\} ▥ s = s$ holds for each sequence s (see page 122) and $\{\} ◫ b = b$ holds for each bag b (see page 94).

Inductive step: To prove the inductive step we have to prove that

$$\forall\, ProgramStateD;\ procs : \mathbb{F}\, Process;\ p : Process \bullet \forall\, ts : TSsD \bullet \\ \quad items((\{p\} \cup procs) \mathbin{\text{▥}} ts.PendTempD) = (\{p\} \cup procs) \mathbin{\text{◫}} items\ ts.PendTempD \qquad (1)$$

holds on the assumption that the *inductive hypothesis* is true:

$$\forall\, ProgramStateD;\ procs : \mathbb{F}\, Process \bullet \forall\, ts : TSsD \bullet \\ \quad items(procs \mathbin{\text{▥}} ts.PendTempD) = procs \mathbin{\text{◫}} items\ ts.PendTempD$$

We prove this by showing that both sides of (1) are equal. The left-hand side of (1) is:

$$\begin{aligned} & items((\{p\} \cup procs) \mathbin{\text{▥}} ts.PendTempD) \\ &\quad = items(\{p\} \mathbin{\text{▥}} (procs \mathbin{\text{▥}} ts.PendTempD)) && \text{[by definition of ▥ (page 122)]} \\ &\quad = \{p\} \mathbin{\text{◫}} items(procs \mathbin{\text{▥}} ts.PendTempD) && \text{[by inductive hypothesis]} \\ &\quad = \{p\} \mathbin{\text{◫}} (procs \mathbin{\text{◫}} items\ ts.PendTempD) && \text{[by inductive hypothesis]} \\ &\quad = (\{p\} \cup procs) \mathbin{\text{◫}} items\ ts.PendTempD && \text{[by definition of ◫ (page 94)]} \end{aligned}$$

This equals the right-hand side of (1). □

9.4.2 Satisfaction of the Fairness Properties

We have to show that the fairness properties (Sect. 8.7.5) are satisfied. First, we prove that processes which are pending for templates are selected in a weakly fair way. The fairness property for our concrete state space is defined as follows:

$$\forall\, ts : TSsD;\ pp : Pending \mid pp \in \operatorname{ran} ts.PendTempD \bullet \\ \quad \Box\Diamond(\exists\, ots : TSsD \mid ots.IdD = ts.IdD \bullet (\exists\, tup : \operatorname{dom} ots.TuplesD \bullet tup \ \mathsf{Matches}\ pp.temp)) \\ \qquad \Rightarrow \Diamond(\exists\, ots : TSsD \mid ots.IdD = ts.IdD \bullet pp \notin \operatorname{ran} ots.PendTempD)$$

See page 111 for the definition of this fairness property for the abstract state space. We sketch the proof for the satisfaction of this property by the operations of our design:

Proposition: The processes which are pending for templates are selected in a weakly fair way by the operations *DepositOKD*, *FetchNoMatchD*, and *MeetNoMatchD* of our design.

Sketch of proof: We distinguish between `meet` templates without `into`s and *other* templates (`meet` templates with `into`s and `fetch` templates):

- `meet` templates without `into`s are selected without consideration of their position in the *PendTempD* queue, thus they will be selected at the first time a matching tuple is deposited: *pp* is not in ran *ots.PendTempD* in the next state after which a matching tuple has been deposited.
- The *PendTempD* queue is managed in FIFO order by the operations *DepositOKD*, *FetchNoMatchD*, and *MeetNoMatchD* (see Fig. 9.1).

  Only templates which match are considered when a new tuple is deposited (see *MatchIntosFetchs* in AddTupleD). The first template in the queue *MatchIntosFetchs* is selected and all templates from the associated process are removed from the *PendTempD* queue.

  Let $n = \#PendTempD$ in any state in the history where $pp \in \operatorname{ran} ts.PendTempD$. Then *pp* has to wait at most *n* times a matching tuple is deposited after this state. Since there is always eventually (infinitely often) a matching tuple for *pp* available (according to the fairness property), *pp* will be selected eventually. Therefore, *pp* is not in ran *ots.PendTempD* in the next state after which it has been selected. □

Weakly fair selection of processes that are pending for full tuple spaces in the concrete state space is defined as follows:

$$\begin{array}{l} \forall\, ts : TSsD;\ b : Process \times APTuple \mid b \in \operatorname{ran} ts.PendFullD \bullet \\ \quad \Box\Diamond(\exists\, ots : TSsD \mid ots.IdD = ts.IdD \bullet IntValueOf\ ots.LimitD > BagSum\ ots.TuplesD) \\ \qquad \Rightarrow \Diamond(\exists\, ots : TSsD \mid ots.IdD = ts.IdD \bullet b \notin \operatorname{ran} ots.PendFullD) \end{array}$$

See page 111 for the definition of this fairness property for the abstract state space. We sketch the proof for the satisfaction of this property by the operations of our design:

Proposition: The processes which are pending for full tuple spaces are selected in a weakly fair way by the operations *FullTSBlockD* and *FetchMatchD* of our design.

Sketch of proof: The *PendFullD* queue is managed in FIFO order by the operations *FullTSBlockD* and *FetchMatchD* (see Fig. 9.2).

Let $n = \#PendFullD$ in any state in the history where $b \in \operatorname{ran} ts.PendFullD$. Then b has to wait at most n times a tuple is fetched after this state. Since there is always eventually (infinitely often) a tuple fetched (i.e., the limit is bigger than the actual sum of tuples), b will be selected eventually. Therefore, b is not in $\operatorname{ran} ots.PendFullD$ in the next state after which it has been selected. □

9.5 Summary

We developed a design for our semantics. We specified the relation between abstract and concrete state space formally in Sect. 9.2, we proved that the concrete operations are correct implementations of the abstract operations, and we proved that the concrete operations satisfy the fairness properties. In this way we demonstrated how to improve the confidence in the correctness of our design. When formal methods are systematically applied to all stages of design and implementation, we increase the confidence that the software is robust and correct. However, we remark that refinement is not sufficiently supported by Z: we had to explicitly rewrite the entire abstract operations to obtain the concrete operations despite the fact that we refined only parts of the schemas. We did not present all refined operations to save space.

Note, however, that an implementation design, which is considered to be correct, cannot be claimed to be fully reliable. A design that has been verified is *not* immune from bugs, although the probability that it contains bugs is very much smaller than if it had not been verified. Writing a proof is somewhat like writing a program, and is subject to error in the same ways that programs are subject to error. Nevertheless, a proof does give us a very high degree of confidence in the correctness of a program, even though it cannot guarantee total reliability. Some chance of failure always remains, no matter how remote. [DeMillo *et al.*, 1979] argue that believing a proof is a social process. A proof for a refinement step is a *message* to the community, which says why we *believe* that it is correct. Therefore, our goal is not to design a fully reliable implementation — we even do not believe that this would be possible in practice. Our goal is to gain high confidence in the design for the implementation.

In our design, the tuple-space manager selects matching templates according to their position in the sequences (the queues). Also multiple queues for pending processes, separated with respect to possible matching tuple sets, may be useful. The semantics does not restrict us to a particular design (as long as it is a valid interpretation of the semantics). An implementation according to the design given in this chapter restricts the nondeterminism possible in the semantics, but still prevents starvation.

The formal specification of Chap. 8 precisely defines the duties for the implementor. The implementation design of this chapter is the first step for implementing ProSet-Linda. It is the basis for the prototype implementation of Chap. 10.

Chapter 10

A Prototype Implementation

As discussed in Sect. 1.2.2, after specifying the formal semantics of a proposed system a prototype should be build rapidly to validate the formal specification. Such a prototype enables us to *test* the specification concerning its adequacy. The execution and presentation of the prototype is then called *animation* of the specification [Diller, 1990]. We present in this chapter a prototype implementation of the runtime system for ProSet-Linda in ProSet itself. This implementation is directly derived from the formal specification.

We will briefly discuss the implementation of the compiler in Sect. 10.1, and present the prototype implementation for the tuple-space management in Sect. 10.2.

10.1 The Compiler

ProSet is compiled and not interpreted. The compiler construction system Eli [Gray *et al.*, 1992] is the central tool for implementing ProSet. Eli integrates off-the-shelf tools and libraries with specialized language processors to provide a system for generating complete compilers quickly and reliably. The first prototype of the compiler translates a subset of ProSet into SETL2 [Snyder, 1990]. The translation with Eli is documented in [Hasselbring, 1991c]. It is quite easy to translate ProSet into SETL2, since both languages are successors to SETL [Schwartz *et al.*, 1986]. The production compiler compiles ProSet into C. This compiler is discussed in [Doberkat *et al.*, 1992c]. The front-end is essentially identical for both compilers, but the code production differs considerably. In the remainder of this chapter we will use the term *compiler* as a synonym for *prototype of the compiler*.

The subset which is translated by the compiler includes the tuple-space operations and the library functions for handling multiple tuple spaces. We do not intend to present the compiler implementation here. This has been done in [Hasselbring, 1991c]. Instead we present the prototype implementation for the runtime system of the tuple-space operations and the library functions for handling multiple tuple spaces in Sect. 10.2 viz., the tuple-space management.

SETL2 does not support any form of parallelism. Therefore, we cannot compile the `||` operator for process creation. This is a limitation in our prototype implementation, which is caused by the limited capabilities of the available tools. However, despite this restriction the prototype enables us to test *essential* features of ProSet-Linda: the tuple-space operations and the library functions for handling multiple tuple spaces. Prototypes usually model only essential features of proposed systems (Chap. 2). The absence of parallelism simplifies our prototype implementation of tuple spaces considerably, since it makes no sense to block processes: no other process can enable a blocked process.

```
module TupleSpaceManager
  export CreateTS, ExistsTS, ClearTS, RemoveTS, Deposit, Fetch, Meet, Formal;

  visible TSs, ActiveProcs,
          IsFormal;
begin  -- the initialization of module instances:
  TSs := {};
  ActiveProcs := {};
  IsFormal := newat();    -- indicates formals to Fetch and Meet

  ...
end TupleSpaceManager;
```

Figure 10.1: The main part of the ProSet implementation for the tuple-space management. The procedures in the subsequent figures of this chapter belong to this module.

10.2 The Tuple-Space Management

The prototype implementation for the tuple-space management is implemented in ProSet and has been translated into SETL2 by the compiler. Therefore, tuple-space operations in ProSet programs are translated by the compiler into calls to those procedures which are part of the tuple-space management.

Fig. 10.1 displays the main part of the tuple-space management module. The program state is represented by a set of tuple spaces and a set of active processes corresponding to the formal specification of *ProgramStateD* on page 119. The atom `IsFormal` indicates formals to `Fetch` and `Meet` (see below). Variables, which are declared as `visible` on the top level of modules, are *static variables* for instances of these modules. These variables are visible to each procedure within the corresponding module, and they are only visible to the encapsulated procedures. See Sect. 4.6 for an introduction to modules in ProSet.

A ProSet-Linda program has to *instantiate* the tuple-space management before using tuple-space operations:

```
instantiate TupleSpaceManager end instantiate;
```

The procedures in the `export` list of `TupleSpaceManager` are available after instantiation. Procedures which are not listed in the `export` list are local to the module. ProSet's module concept is mapped to SETL2's packages by the compiler: the instantiation of a module does not return a module instance as it is the case in ProSet (see Sect. 4.6). The exported procedures may be called after instantiation as if they were defined in the instantiating program. It is not necessary to prefix the names of exported procedures with the name of a module instance (which is the case in ProSet). See [Snyder, 1990] for a description of packages in SETL2.

10.2.1 Handling Multiple Tuple Spaces

The four functions for handling multiple tuple spaces are displayed in Fig. 10.2. They belong to the tuple-space management module.

`CreateTS`: First, the given `limit` is checked. If the `limit` is not an integer or the undefined value, the exception `type_mismatch` is raised via `escape` (see Sect. 4.4). Since SETL2 does not support exception handling, the compiler translates `escape` statements into statement sequences which print out appropriate messages and stop program execution.

```
procedure CreateTS (limit);
begin
  if limit /= om and type limit /= integer then
    escape type_mismatch();
  end if;
  newTS := [newat(), limit, {}];
  TSs with:= newTS;
  return newTS(1);  -- the tuple-space identity
end CreateTS;

procedure ExistsTS (tsid);
begin
  if type tsid /= atom then
    escape type_mismatch();
  end if;
  if exists ts in TSs | (ts(1) = tsid) then
    return true;
  else
    return false;
  end if;
end ExistsTS;

procedure ClearTS (tsid);
begin
  CheckTSID (tsid);

  myts := arb {ts: ts in TSs | ts(1) = tsid};
  TSs less:= myts;  -- remove the tuple space
  myts(3) := {};    -- delete the tuples
  TSs with:= myts;  -- insert the empty tuple space
end ClearTS;

procedure RemoveTS (tsid);
begin
  CheckTSID (tsid);

  TSs -:= {ts: ts in TSs | ts(1) = tsid};
end RemoveTS;

procedure CheckTSID (tsid);  -- auxiliary function to check tuple-space identities
begin
  if type tsid /= atom then
    escape type_mismatch();
  end if;
  if not (exists ts in TSs | (ts(1) = tsid)) then
    escape ts_invalid_id();
  end if;
end CheckTSID;
```

Figure 10.2: The library functions for handling multiple tuple spaces.

Tuple spaces are implemented through tuples with three components. The position of these components corresponds to the position of the components in the schema *TupleSpaceD* on page 118. The first component is the tuple-space identity. The predefined function `newat` returns a fresh atom. The second component is the specified limit. The third component of a tuple space implements the multiset of tuples in our prototype implementation. As we see in Fig. 10.2, this multiset is initially empty. Because ProSet does not directly provide multisets, we model multisets of tuples through maps from tuples to counts. Z supports multisets in the same way (see Chap. 8 and Appendix B).

It is not necessary to implement the components *PendTempD* and *PendFullD* of *TupleSpaceD*, because our prototype implementation does not support parallelism and blocked processes (as indicated in Sect. 10.1).

`ExistsTS`: This function simply checks whether a given atom is a valid tuple-space identity. The parameter has to be an atom.

`ClearTS`: This function removes all tuples from a specified tuple space. The auxiliary function `CheckTSID` first checks whether the parameter is a valid tuple-space identity. ProSet's unary operator `arb` returns an arbitrary element from a set. The set {`ts: ts in TSs | ts(1) = tsid`} in `ClearTS` will always have exactly one element because `CheckTSID` would have raised an exception otherwise (see Fig. 10.2). First, the entire tuple space is removed from `TSs`. The third component (the multiset of tuples) is then cleared, and the resulting empty tuple space is inserted into `TSs`.

`RemoveTS`: This function removes the entire tuple space from `TSs`.

10.2.2 Tuple-Space Operations

The tuple-space operations are mapped to the respective procedures `Deposit`, `Fetch`, `Meet`, and `Formal` which belong to the tuple-space management.

Depositing Tuples

A `deposit` operation is directly translated into a call to the procedure

```
  procedure Deposit (tup, tsid);
```

of the tuple-space management, where `tup` is the tuple to be deposited and `tsid` specifies the tuple space. Fig. 10.3 displays the procedure `Deposit`. First, the given tuple and the tuple-space identity are checked. The specified tuple space is selected in the same way as in `ClearTS` (Sect. 10.2.1). Then it is checked whether the selected tuple space is full. The compiler translates the `signal` statement (Sect. 4.4) in the same way as it translates the `escape` statement into a statement sequence which prints out an appropriate message and stops the program execution (resuming is not supported). If no exceptional situations are discovered, the tuple is added to the tuple space. See Fig. 10.3 for details.

Fetching Tuples

A `fetch` operation is translated into calls to the procedure

```
  procedure Fetch (actuals, rw formals, Condition, tsid);
```

of the tuple-space management, where `actuals` is a tuple representing the actuals of a template and `formals` is a tuple representing the formals of a template. `actuals` contains the template's actuals in the corresponding positions and the atom `IsFormal` (Fig. 10.1) otherwise. `formals` contains the

```
procedure Deposit (tup, tsid);
begin
  if type tup /= tuple then
    escape type_mismatch();
  end if;
  CheckTSID (tsid);

  myts := arb {ts: ts in TSs | ts(1) = tsid};

  if myts(2) /= om and BagSum(myts(3)) >= myts(2) then
    signal ts_is_full(); -- resuming is not supported
  end if;

  TSs less:= myts;  -- remove the tuple space
  myts(3)(tup) := if myts(3)(tup) = om then 1
                                        else myts(3)(tup)+1
                  end if;
  TSs with:= myts;  -- insert the tuple space with the deposited tuple

  procedure BagSum (ts); -- computes the sum of elements in a multiset
  begin
    sum := 0;
    for pair in ts do
      sum +:= pair(2);   -- add the count
    end for;
    return sum;
  end BagSum;
end Deposit;
```

Figure 10.3: Depositing tuples.
See Fig. 10.2 for `CheckTSID`. The procedure `BagSum` is the counterpart of *BagSum* in the formal specification (page 102).

l-values of the template's formals in the corresponding positions and a predefined dummy *l*-value otherwise (see below). `Condition` is a λ-expression which computes the template condition and `tsid` specifies the tuple space. Fig. 10.4 displays the procedure `Fetch`.

Let us first discuss the translation of a specific example before explaining the procedure `Fetch` proper. The example for the `fetch` operation from page 58:

```
fetch ( "name", ? x |(type $(2) = integer) ) => put("Integer fetched");
  xor ( "name", ? x |(type $(2) = set) )     => put("Set fetched");
   at TS
 else put("Nothing fetched");
end fetch;
```

is translated by the compiler into a SETL2-`CASE` statement:

```
CASE
  WHEN Fetch ([ "name", Formal() ], [L_NF, x ],
       LAMBDA (L_TUP); RETURN ( TYPE (L_TUP(2)) = "INTEGER" ); END LAMBDA, TS ) =>
           PRINT ( "Integer fetched" );
  WHEN Fetch ([ "name", Formal() ], [L_NF, x ],
       LAMBDA (L_TUP); RETURN ( TYPE (L_TUP(2)) = "SET" ); END LAMBDA, TS ) =>
           PRINT ( "Set fetched" );
  OTHERWISE =>
           PRINT ( "Nothing fetched" );
END CASE;
```

The produced SETL2-code and the procedure `Fetch` work together as follows: the `Fetch` procedure is called once for each template. `Fetch` returns `true` when it finds a matching tuple, otherwise `Fetch` returns `false`. This `CASE` statement is similar to a guarded command. Each of the expressions after the keyword `WHEN` is evaluated in an unspecified order. When one is found which evaluates to `true`, the associated statement list is executed. If none of the expressions evaluate to true then the `OTHERWISE` clause is executed. Note that a `fetch` operation does not block when no matching tuple is available: if no `else` statements are specified, the compiler inserts an `OTHERWISE` clause to print the message "No matching tuple found".

The procedure `Formal` returns the atom `IsFormal` to indicate a formal within the parameter `actuals` in the corresponding position (see Fig. 10.4). The parameter `formals` is a `rw`-parameter (read-write): read-write parameters are transmitted *call by value/result* (see Sect. 4.1). On entry to the procedure the formal parameter is initialized with the actual parameter. When the procedure terminates, the current value of the formal parameter is assigned to the *l*-value of the `rw`-parameter. The parameter `formals` is a *multiple l-value* (a tuple of *l*-values): a tuple value has to be assigned to `formals` (see Sect. 4.1). The tuple components are then assigned to the individual *l*-values according to their position in the assigned tuple value.

The parameter `formals` is the inverse to `actuals`: for each actual it contains the dummy *l*-value `L_NF` in the corresponding position. The template condition is enclosed in a λ-expression. Such λ-expressions yield first-class functions which can be executed [Snyder, 1990]. The procedure `Matches`, which is called by `Fetch`, will call these first-class functions with possibly matching tuples as actual parameters (see below). The `$`-expression is translated into the formal parameter `L_TUP` (see the previous example). `L_TUP` and `L_NF` are keywords for the compiler to avoid conflicts with user-defined names: the use of `L_TUP` and `L_NF` by the programmer is rejected by the compiler.

SETL2's `TYPE` operator returns a string and not an atom as it is the case in ProSet. The compiler translates the predefined type atoms accordingly and emits appropriate warning messages when ProSet's `type` operator and the predefined type atoms are used.

Fig. 10.4 displays the procedure `Fetch`. At first the given tuple-space identity is checked. The specified tuple space is selected in the same way as it is done in `ClearTS` (Sect. 10.2.1). If no exceptional

```
procedure Fetch (actuals, rw formals, Condition, tsid);
begin
  CheckTSID (tsid);

  myts := arb {ts: ts in TSs | ts(1) = tsid};

  tup := Matches (myts(3), actuals, Condition);

  if tup = om then
    formals := formals;       -- restore the formals
    return false;             -- no matching tuple found
  else
    formals := tup;           -- assign the formals
    TSs less:= myts;          -- remove the tuple space
    if myts(3)(tup) = 1 then
      myts(3) lessf:= tup;    -- remove the (tuple, count) pair
    else
      myts(3)(tup) -:= 1;     -- decrease the count
    end if;
    TSs with:= myts;          -- insert the tuple space without the fetched tuple
    return true;
  end if;
end Fetch;

procedure Matches (tuples, actuals, Condition);
begin
  for pair in tuples do
    if #pair(1) = #actuals and Condition(pair(1)) then
      DoesMatch := true;                   -- might be a matching tuple
      for i in [1..#pair(1)] do
        if actuals(i) /= IsFormal then     -- formals match unconditionally
          if pair(1)(i) /= actuals(i) then
            DoesMatch := false;            -- an actual does not match
          end if;
        end if;
      end for;
      if DoesMatch then
        return pair(1);                    -- return the matching tuple
      end if;
    end if;
  end for;
  return om;                               -- no matching tuple found
end Matches;

procedure Formal ();  -- returns the atom IsFormal to indicate a formal
begin
  return IsFormal;
end Formal;
```

Figure 10.4: Fetching tuples.
See Fig. 10.1 for `IsFormal` and Fig. 10.2 for `CheckTSID`. The operation `f lessf:= x` deletes from the map `f` all pairs in which the first component is equal to `x`.

situations are discovered, the procedure `Matches` is called to find a matching tuple in the multiset of tuples. If no matching tuple is found, the formals are restored and `false` is returned. If a matching tuple is found, the formals are assigned the corresponding tuple fields and the tuple is removed from the tuple space. See Fig. 10.4 for details.

The procedure `Matches` checks for each tuple in a multiset of tuples whether it matches the given template. `Matches` does not need the parameter `formals` of `Fetch`, since formals match unconditionally. It calls the λ-expression `Condition` to compute the template condition. The procedure `Matches` is the counterpart of Matches in the formal specification (page 92).

Meeting Tuples

A `meet` operation is translated into calls of the procedure

```
  procedure Meet (actuals, rw formals, intos, Condition, tsid);
```

of the tuple-space management, where the formal parameters `actuals`, `formals`, `Condition`, and `tsid` are used in the same way as they are used in the procedure `Fetch`. The additional parameter `intos` is a tuple containing the undefined value where no `into` is specified in the given template and a λ-expression where an `into` is specified in the corresponding position. This tuple is empty when no `into`s are specified, because trailing `om`s are removed from tuples in ProSet (see Sect. 4.1). Fig. 10.5 displays the procedure `Meet`.

Let us first discuss the translation of a specific example before explaining the procedure `Meet` proper. The following `meet` operation

```
meet ( "name", ? x into $(2)+1 |(type $(2) = integer) ) => put("Integer changed");
   at TSid
 else put("Nothing met");
end meet;
```

is translated by the compiler into a SETL2-`CASE` statement very similar to the translation of `fetch` operations:

```
CASE
  WHEN Meet ([ "name", Formal() ], [L_NF, x ],
       [ OM, LAMBDA (L_TUP); RETURN (L_TUP(2) + 1); END LAMBDA ],
       LAMBDA (L_TUP); RETURN ( TYPE (L_TUP(2)) = "INTEGER" ); END LAMBDA, TSid ) =>
    PRINT ( "Integer changed" );
  OTHERWISE =>
    PRINT ( "Nothing met" );
END CASE;
```

The `into` expressions are enclosed in λ-expressions similar to the template condition. Fig. 10.5 displays the procedure `Meet`. At first the given tuple-space identity is checked. The specified tuple space is selected in the same way as it is done in `ClearTS` (Sect. 10.2.1). If no exceptional situations are discovered, the procedure `Matches` is called to find a matching tuple in the multiset of tuples. If no matching tuple is found, the formals are restored and `false` is returned. If a matching tuple is found, the formals are assigned the corresponding tuple fields and it is checked if `into`s are specified. If no `into`s are specified, `Meet` returns `true` and does not remove the tuple. If `into`s are specified, the found tuple is removed from the tuple space. The tuple components, for which `into`s are specified, are replaced by the return values of the corresponding λ-expressions in the parameter `intos`, and then the changed tuple is added via `Deposit`. See Fig. 10.5 for details.

```
procedure Meet (actuals, rw formals, intos, Condition, tsid);
begin
  CheckTSID (tsid);

  myts := arb {ts: ts in TSs | ts(1) = tsid};

  tup := Matches (myts(3), actuals, Condition);

  if tup = om then
    formals := formals;         -- restore the formals
    return false;               -- no matching tuple found
  else
    formals := tup;             -- assign the formals
    if intos = [] then
      return true;              -- do not change the tuple space
    else
      TSs less:= myts;          -- remove the tuple space
      if myts(3)(tup) = 1 then
        myts(3) lessf:= tup;    -- remove the (tuple, count) pair
      else
        myts(3)(tup) -:= 1;     -- decrease the count
      end if;
      TSs with:= myts;          -- insert the tuple space without the removed tuple
      for i in [1..#intos] | intos(i) /= om do
        tup(i) := intos(i)(tup); -- change the tuple component
      end for;
      Deposit (tup, tsid);       -- deposit the changed tuple
      return true;
    end if;
  end if;
end Meet;
```

Figure 10.5: Meeting tuples.
See Fig. 10.2 for `CheckTSID`, Fig. 10.3 for `Deposit`, and Fig. 10.4 for `Matches`.

10.3 Summary

We presented a prototype implementation for ProSet's tuple-space management. This prototype enables us to *test* the specification concerning its adequacy. We tested, among other test programs, the examples of Chap. 6. Because we cannot compile the `||` operator, we had to make some modifications. This is due to the limited capabilities of the available tools. For instance, in the solution for the queens' problem (Sect. 6.2) we call only one worker, and in the solutions for parallel matrix multiplication (Sect. 6.3) and the traveling salesman problem (Sect. 6.4) we call the workers sequentially instead of spawning them via `||`. However, the programs produce the same results because the workers work independently of each other. Therefore, we have the opportunity to test essential features of ProSet-Linda early in the process of software construction. As discussed in Chap. 2 prototypes usually do not model *all* features of a proposed system.

The main part of the implementation work for the prototype was not the implementation of the tuple-space management. It was quite easy to write the ProSet procedures with the formal specification on the desk. The main work was the implementation of the compiler. Experience with using Eli for implementing the compiler is documented in [Hasselbring, 1991c; Doberkat *et al.*, 1992c].

After the work for this thesis was completed, we translated the tuple-space management with the production compiler into C. Accordingly, the production compiler translates the tuple-space operations into calls of the corresponding produced C functions. We extended these C functions to support synchronization for parallel access to the tuple spaces and dynamic process creation with the `||` operator: this second prototype supports parallelism. We actually use the SunOS Lightweight Processes Library [Sun, 1990] to implement process creation and synchronization.

In many current operating systems the *lightweight process* or *thread* has emerged as a useful representation of computational activity. Lightweight processes represent multiple threads of control which share the address space of a single heavyweight process. Lightweight processes usually cooperate closely and frequently with each other and are typically used to implement parts of a program which are best executed concurrently. In operating systems like Unix lightweight processes are provided to heavyweight processes by a library which allows the user to execute functions as lightweight processes. We refer to [Tanenbaum, 1992, Sect. 12.1] for an introduction to lightweight processes.

As we can see, the production-level implementation will be based on the prototype implementation: our prototype is not thrown away. Chapter 11 will discuss some general issues concerning the production-level implementation of ProSet-Linda. Proposed changes in the language design can first be evaluated with the prototype implementation in ProSet before they are incorporated into the production-level implementation in C. This approach allows an evolutionary development.

We do not present proofs for the correspondence between the formal specification and the prototype implementation. Even textbooks on Z do not provide proofs when executable prototypes are developed from Z specifications. For instance, in [Diller, 1990] prototypes for Z specifications were constructed in Miranda and PROLOG. There, the Z specification is "straightforwardly translated into Miranda" (page 218) and the "PROLOG animation of it should be fairly clear" (page 223). No formal proofs are given for the translation of Z specifications into Miranda and PROLOG. Jim Woodcock writes in an introductory tutorial on refinement in Z concerning his development into code: "The development into code is straightforward" [Woodcock, 1991, page 124]. The reasons for not providing proofs for the development into code are that such proofs are usually very complex and do not essentially increase the confidence in the correctness of the development. Since the implementation in a very high-level language — such as ProSet — is straightforward, we already have high confidence in the correspondence between the formal specification and the prototype. Additionally, programming languages like ProSet and SETL2 are not defined with formal semantics for programs in the sense that formal specification languages are defined with formal semantics for specifications. This difference makes it hard (and even impossible in practice) to formally specify the relations between programs and specifications.

Chapter 11

Some General Issues for Implementations of ProSet-Linda

In this chapter we will discuss some general issues concerning implementing ProSet-Linda. Section 11.1 presents a short summary of some existing implementations of Linda variants and Sect. 11.2 sketches implementation techniques. Optimizations and the somewhat unpredictable performance of Linda are discussed in Sects. 11.3 and 11.4, respectively.

11.1 Some Existing Implementations of Linda Variants

This section summarizes some existing implementations of Linda. We do not claim to provide a comprehensive overview. Our intention is to show that Linda has been implemented on a wide variety of parallel architectures and is, therefore, portable across different machine architectures. The implementation techniques applied are discussed in Sect. 11.2. For a survey and a taxonomy of parallel computer architectures we refer the reader to [Flynn, 1966; Duncan, 1990].

Implementations of Linda have been performed on shared-memory multi-processors as well as on distributed memory architectures:

Sequent Balance / Encore Multimax [Carriero, 1987; Char, 1990]
These are shared-memory multi-processors with National Semiconductor processors.

Symmetric multi-processing VAX/VMS [Kane, 1991]
Symmetric multi-processing (SMP) is a bus-based shared-memory model of parallel computation. Each processor in a SMP VAX can independently and equally access all operating system code and system resources. The processors reside physically close to each other, so that they can communicate through shared memory.

S/Net [Carriero and Gelernter, 1986; Carriero, 1987]
This is a bus-based distributed memory machine consisting of Motorola 68000 processors. The bus supports reliable broadcast.

Parwell [Borrman *et al.*, 1989]
This is a distributed memory machine consisting of Motorola 68020/68881 processor pairs which are connected via a hierarchical bus system.

AP1000 [Cohen and Molinari, 1991]
In this distributed memory machine, SPARC processors are arranged in a two-dimensional array. This system is also referred to as a Cellular Array Processor.

Transputer [Dourish, 1989; Ushijima, 1989; Zenith, 1990; Leler, 1990; Clayton *et al.*, 1990; Callsen *et al.*, 1991; Faasen, 1991; Shekhar and Srikant, 1991; Smith, 1992; Trescher *et al.*, 1992]
The Transputer is a RISC-like processor which has four high-speed communication links. A Transputer network is a distributed memory architecture, where the nodes are usually arranged in a grid, giving each node exactly four neighbors.

Hypercubes
Transputers and Cellular Array Processors are arranged in two-dimensional grids. In hypercubes, the processors are connected logically in more than two dimensions. Linda has been implemented on such systems with Motorola 68000 processors [Lucco, 1986], Intel 80386/80387 processor pairs [Bjornson, 1992], and Intel 80860 processors [Bogoch *et al.*, 1990].

Local area networks
Conventional workstations connected by a local area network:

DOS connected through NetBIOS [Bettermann, 1992]

Unix connected through Ethernet [Arango and Berndt, 1989; Fleckenstein and Hemmendinger, 1989; Gelernter and Philbin, 1990; Murakami *et al.*, 1991; Pinakis, 1991; Chiba *et al.*, 1991; Schoinas, 1991; Thomas, 1991; Patterson *et al.*, 1992]

VAX/VMS connected through Ethernet [Leichter, 1989]

Some commercial implementations are reported in [Markoff, 1992]. The public domain implementation POSYBL is discussed in [Stapleton, 1991; Schoinas, 1991].

11.2 Implementation Techniques

In implementations of Linda on shared-memory multi-processors like the Sequent Balance, the Encore Multimax or the SMP VAX it is straightforward to store the tuple space in shared memory. We will only discuss implementations on distributed memory architectures here, since such systems are widely available, scalable and provide high performance for modest costs compared to shared-memory multi-processor systems. *Scalability* means the capability of delivering an increase in performance proportional to an increase size. Shared-memory systems are expensive and scale up only to a few dozen processors, because the contention for access to shared-memory quickly becomes a bottleneck. Conversely, distributed memory architectures scale up to thousands of processors.

Even today's local area networks may be used to execute parallel applications. However, local area networks are problematic hosts for parallel applications, because the communication links are slow compared to real parallel machines such as Transputer networks. At most sites, workstations connected by local area networks are often idle and may, therefore, become a parallel computer system.

On distributed memory architectures, a general problem for implementations of Linda is to provide a map from the virtual shared memory model to physical distributed memory architectures. A central store needs a complex kernel (tuple-space manager), which may become a bottleneck. A replicated store may cause consistency problems. Therefore, efficient and reliable implementations of Linda on physical distributed memory architectures are in general a great challenge for the implementor. Implementation techniques for physical distributed memory architectures range from ones where the tuple space is replicated on each node to those where each tuple resides on exactly one node. The implementation techniques may be classified as follows:

1. Central store with server process
2. Replication of the entire tuple space at each node
3. Distribution of the tuple space over the net with unique copies of each tuple

4. Mixture of these techniques

We discuss these items in turn now:

1. In this implementation technique, the central store may very soon become both a computational and a communicational bottleneck. This simple implementation technique is not discussed further here.

2. In this implementation technique, broadcasting of tuples is usually applied when tuples are deposited [Carriero, 1987]. For such a replication of the entire tuple space at each node the system should provide broadcasting features. Reading of tuples is cheap with this technique, since no exchange of messages is necessary; as soon as a matching tuple is found on the local node, it is readily available for the reading process. Fetching of tuples triggers a local search for a matching tuple. If one is found, the originating node sends a broadcast "delete the found tuple" to the other hosts. If each host responds "ok, the found tuple has been deleted", the tuple will be fetched. If at least one host responds "the found tuple has already been fetched by another one", the tuple cannot be fetched, and a *delete protocol* has to be applied to retain consistency. The delete protocol has to satisfy two requirement:

   - All involved nodes must receive the delete message.
   - If many processes attempt to delete the same tuple simultaneously, only one will succeed.

   We refer to [Carriero and Gelernter, 1986] for a discussion of some delete protocols for the S/Net. In [Chiba *et al.*, 1991], some consistency protocols for a replication on local area networks are discussed.

   If the local search triggered by an `in` operation turns up no matching tuple, all newly-arriving tuples are checked until a match occurs, at which point the matched tuple is deleted and returned as before.

   A major problem with replication of tuples is to guarantee the consistency of replicated data in a distributed environment. A possible solution is the use of an atomic broadcast protocol (also used in Orca, see Sect. 3.2.8). This protocol ensures that all operations are performed at all replicas in the same order, and in addition that operations of processes are performed in the very same order issued. However, atomic broadcast protocols are not supported on all distributed memory machines.

   Another problem with replication is that the tuple space takes up a lot of memory on each of the nodes in the network. The space used will be n times as much as necessary, where n is the number of nodes in the network.

3. This implementation technique is in some sense the inverse of technique 2: templates are usually broadcast and tuples remain on their nodes of origin until they are explicitly asked for. This technique is applied in [Leichter, 1989].

   The implementation on hypercubes discussed in [Bjornson, 1992] distributes the tuple space, but does not use broadcast of tuples or templates to all or parts of the network, since their hypercube system does not support reliable broadcast. The basic idea in [Bjornson, 1992] is to have as short communications distances between communicating processors as possible. The strategy taken for the implementation of the tuple space is called *distributed hashing*. Here both, tuples and templates, are sent to a so-called *rendezvous node*, chosen by a hash function, which is applied on the tuple and template contents. Tuples and templates that are likely candidates for matching are guaranteed to be sent to the same node, where matching takes place. When a matching tuple/template pair is found, the template is discarded and the matched tuple is sent back to the requesting node. In order to determine the address of the rendezvous node, a hash function is used. It uses information from compile-time analysis to calculate physical network addresses from tuple and template contents. Therefore, tuple storage is based on a distributed table implemented in terms of the tuple classification worked out at compile-time. A similar technique is applied in [Cohen and Molinari, 1991] for an implementation on the Cellular Array Processor AP1000.

In [Pinakis, 1991], tuple space is partitioned so that a set of *tuple types* is managed by each host in a local area network. Processes determine the location of tuples of a particular type by contacting *type-server* processes, which are executing on each host in the network. The tuple space is partitioned at runtime. Each type-server in the system executes a locking and agreement protocol which ensures that if many processes wish to simultaneously create a tuple type, only one succeeds and all contending hosts receive the same information of the tuple type's location.

4. This implementation technique is, e.g., applied in [Patterson *et al.*, 1992], where tuples are replicated among so-called *subspaces* of tuple space and not in the entire system. In [Xu and Liskov, 1989], fault-tolerance is achieved by uniform replication of tuple space on a small subset of available processors in a network. Fault-tolerance is guaranteed with respect to node crashes and network failures. See also Sect. 7.14 for a discussion on this subject. In [Ahuja *et al.*, 1988], tuples are replicated on $\sqrt{n}$ nodes on an n node network. This is discussed in Sect. 11.3.3. A similar technique is applied in [Faasen, 1991] for an implementation on a Transputer system.

We conclude that a distribution of tuple space over the nodes in a parallel system in one form or another is the most promising implementation technique on distributed memory architectures for Linda's tuple space. This is due to several reasons. First, memory is saved and second, the consistency problems imposed by a replication of tuple space are absent. Furthermore, any Linda implementation that can scale to large machines *must* distribute tuple space, so as to avoid node contention. This distributes the cost of handling tuple operations across all nodes in the system. The remaining problem is *how* to distribute the tuple space. Multiple tuple spaces, as they are supported in ProSet, provide an immediate way to distribute the tuple spaces.

11.3 Optimizations

The following subsections discuss the optimization of the performance of Linda implementations.

11.3.1 Partitioning the Tuple Space

Especially in the Yale Linda Implementation extensive compile-time optimizations for C-Linda programs are applied. It is proposed in [Carriero and Gelernter, 1990b] to implement each tuple-space operation as a family of operations in the runtime library. The compiler maps each user-level tuple-space operation onto the cheapest available member of the appropriate family. A default runtime library which implements the general operations with the full functionality is taken, and then replaced by making use of specific information about each call and global knowledge about tuple-space access in a program. The compile-time analysis detects cases in which the full functionality of tuple storage and matching is not needed, and replaces the runtime library to handle common special cases efficiently.

The runtime library is replaced on a per-tuple-space-operation basis. It is intended to accomplish as much as possible of the work involved in executing a tuple-space access before runtime. The compiler checks whether or not runtime matching is needed. If consistently-used constant fields can be *pre-matched* at compile-time, the general operations are replaced by direct message-passing operations, which do not apply any runtime matching. If runtime matching is needed, a key (tuple field) for hashing is searched. Some multiple-key search methods for Linda are discussed in [Peskin and Segall, 1991].

This approach involves constructing partitions of the global tuple space. The construction is based on an examination of the structure and type of tuples found within a C-Linda program. Partitions are constructed by semantic analysis and from type declarations. Tuple-space operations whose tuple arguments correspond in type and length get placed within the same partition; the representation of individual partitions is therefore *customized* to the structure of the tuples that occupy it. Each partition can then be managed by a separate tuple-space manager. This increases the degree of concurrency in the system, because operations working on different partitions can safely proceed simultaneously. It also decreases the amount of runtime searching the system has to do, since only tuples which possibly match a specific operation need to be examined. In [Clayton *et al.*, 1990], the tuple space is partitioned into hierarchical tuple groups.

We remark that an important disadvantage of the approach taken in the Yale Linda Implementation is that separate compilation is not supported because the compiler has to know all the tuple-space operations which occur in a program at compile-time.

In the present proposal concerning ProSet, multiple tuple spaces already provide a partitioning of tuple space on the language level: the tuple space is partitioned as the programmer says and sees it. No compile-time analysis with respect to partitioning the tuple space is needed.

11.3.2 In-place Updates

A common sequence of operations, such as

```
in ( "counter", ?  i );
out ( "counter", i+1 );
```

can be handled by directly locking and incrementing the value stored in the tuple space instead of removing and re-inserting it. However, as this special case is in general not easy to find for a compiler, it is rarely possible to replace such `in/out` sequences by in-place-update operations. In ProSet, the `meet` operation already provides this functionality (Sect. 5.4.3).

11.3.3 Hardware Support

Hardware support for implementations of Linda has been considered. It is proposed in [Ahuja *et al.*, 1988] that a co-processor takes care of implementing the tuple-space operations. This system uses replication of tuples by introducing tuple/template *beams*. If the net of nodes is rectangular, tuples coming from a node are broadcast to all nodes in the same row as itself, and templates are broadcast to all nodes in the same column as itself. Therefore, broadcasting is done to subsets of the nodes in the system and not to each node in the system. Tuples are replicated on $\sqrt{n}$ nodes on an n node network. When searching for a match, the tuple- and template-beams will cross in exactly one node. The special co-processor takes care of matching and distribution of tuples in the net.

We do not consider specific hardware support for implementing ProSet-Linda because portability is more important for prototyping than high performance on specific hardware.

11.3.4 Data Structure Selection

The semantic analysis for Scheme-Linda in [Jagannathan, 1991] is based on an inference engine that statically computes the set of tuples (and their structural attributes) which may occupy any given tuple space. The result of this inference procedure can be used to customize the representation of individual tuple spaces.

Such an analysis may also be useful in ProSet to select appropriate data structures for individual tuple spaces according to their use and content. However, this is subject for further research and beyond the scope of this thesis.

11.4 The Unpredictable Performance of Linda

A crucial problem of Linda is its unpredictable performance which in part is due to the fact that many different strategies are used to distribute the tuple space [Bal, 1992]. However, this is always the case for implementations of virtual shared memory models on physical distributed memory architectures, if no specific assertions are made for the mapping. Thus the unpredictable performance is not a problem unique to Linda.

Linda does not provide any idea on how it is implemented. Conversely, e.g., in Orca (see Sect. 3.2.8), only one implementation technique has been applied: replication of shared objects, either full replication on all processors, or partial replication on a subset of the available processors based on runtime statistics with primary copies of shared objects [Bal *et al.*, 1992]. Orca intends to make read operations cheap and write operations more expensive, so the user has at least some idea of the relative costs. In Linda one does not know anything about replication and distribution before one knows which compiler is in use. Note, however, that a new implementation of Orca does not replicate all objects [Bal and Kaashoek, 1993]. The decision to replicate objects is based on an integration of compile-time and run-time analysis.

An approach to tackle the problem of the unpredictable performance might be the use of multiple tuple spaces. In ProSet, one could introduce the option to specify on creation that a tuple space has to be replicated:

```
TS := CreateReplicatedTS (limit);
```

This would make read operations cheap and write operations more expensive for *this* freshly created tuple space. Also, the compiler could decide to replicate appropriate tuple spaces. The default would be to distribute tuple spaces.

However, note that for a prototyping language such as ProSet runtime efficiency is not of paramount importance. Efficiency in rapid development and change of prototypes is the primary concern. Therefore, the unpredictable performance of Linda is a minor problem for ProSet. It is surpassed by the flexibility of Linda's tuple-space communication, which is the motivation for combining set-oriented prototyping with generative communication.

11.5 Summary

The implementation design of Chap. 9 is the first step for implementing ProSet-Linda. The prototype implementation of Chap. 10 is used to check the correspondence between informal requirements and formal specifications through testing and experimentation. This last chapter of Part III discusses some general issues concerning implementing ProSet-Linda. The production-level implementation of ProSet-Linda is based on the prototype implementation. A detailed discussion of the production-level implementation is beyond the scope of this thesis and the subject for future research (see also Chap. 13).

Part IV

Résumé and Outlook

Chapter 12

Résumé

The goal of our research is to design a tool for prototyping parallel algorithms. We construct this tool in a somewhat unconventional way: the informal specification is followed by a formal specification, which serves as the basis for a prototype implementation before the production-level implementation is undertaken.

We evaluated several high-level parallel programming models concerning their suitability for prototyping parallel algorithms in a set-oriented language. This evaluation led us to our approach to parallel programming where the concept for process creation via Multilisp's futures is adapted to set-oriented programming and combined with the concept for synchronization and communication via tuple space. The basic Linda model is enhanced with multiple tuple spaces, the notion of limited tuple spaces, selection and customization for matching, specified fairness of choice, and the facility for changing tuples in tuple space. It is fairly natural to combine set-oriented programming with generative communication on the basis of tuples, as both models, ProSet and Linda, provide tuples.

Our goal is to make parallel program design easier through prototyping parallel algorithms. The high level of ProSet's constructs for parallel programming enables us to rapidly develop prototypes of parallel programs and to experiment with parallel algorithms. The subject of this thesis is the construction of prototypes and not the transformation of prototypes into production-quality programs. Therefore, we consider only the early phases in the process of software construction.

A formal specification of ProSet-Linda has been presented by means of the formal specification language Object-Z. The specification of the formal semantics of generative communication in ProSet led us to the recognition of several omissions and imperfections in our previous informal specification. The main advantage of using a formal specification lies in subsequent development steps for the implementation. Formal development involves some sort of transformation/refinement. Refinement is usually regarded as a step-wise approach, possibly with proof obligations for each step. This method of development results for the developer in a higher level of confidence in the end product.

We refined our formal specification into an implementation design and specified formally the relation between the abstract state space of the specification and the more concrete state space of the implementation design. We proved that the concrete operations are correct implementations of the abstract operations and that the concrete operations satisfy the fairness properties. Consequently, we gained high confidence in the correctness of our implementation design. We implemented a prototype from the formal specification and discussed some general issues concerning the implementation of ProSet-Linda. The prototype allows immediate validation of the specification by execution. It is not possible to check the correspondence between informal requirements and formal specifications formally by verification. This situation suggests the following cyclic process for requirements analysis:

$$\begin{tikzcd}
\text{informal requirements} \arrow[r] & \text{formal specification} \arrow[d] \\
\text{prototype animation and evaluation} \arrow[u] & \text{prototype implementation} \arrow[l]
\end{tikzcd}$$

When the requirements reach a stable state, the next phases in the process of software production may begin. The prototype enables us to avoid the large time lag between specification of a system and its validation in more traditional models of software production. The next phases should be based on the formal specification and on the prototype implementation. When a system's requirements change after delivery, it is more appropriate to base changes in the maintenance phase on the flexible prototype instead on the optimized implementation. We refer to [Ghezzi *et al.*, 1991, Chapter 7] for an introduction to software production process models.

ProSet and Z appear to be a good combination for software engineering in general. Because of the similarities between ProSet and Z it may be a good idea to build an executable prototype — derived from a specification written in Z — in ProSet. There exist some other approaches to animation of Z and VDM specifications with PROLOG [Ekambareshwar and Downs, 1989; Diller, 1990; West and Eaglestone, 1992] and with functional languages such as Miranda [Diller, 1990] or ML [O'Neill, 1992]. It has also been proposed to make subsets of Z executable [Valentine, 1992; Gimnich and Ebert, 1992]. For a debate on whether specifications themselves should be executable see [Hayes and Jones, 1989; Fuchs, 1992].

Set-oriented programming techniques may be a more adequate choice for constructing prototypes from Z specifications than techniques from functional or logic programming. ProSet is a procedural language which also contains a Pascal-like subset that facilitates prototyping by allowing a program to be refined into successively finer detail while staying within the language: it is a wide-spectrum language [CIP, 1985]. These features allow us to systematically transform prototypes into production-quality products. For functional or logic languages there is a somewhat wider gap to bridge to arrive at a production-level program. Production-level programs are usually written in procedural languages like C. However, there exist some significant differences between ProSet and Z:

- ProSet is weakly typed, whereas Z is strongly typed.
- ProSet programs are executable prototypes, whereas Z specifications are not executable.
- ProSet only supports finite sets, whereas Z also supports infinite sets.

An executable language is by definition more restricted in expressive power than a non-executable one, because its functions must be computable and are defined over domains with finite representations [Wing, 1990].

Chapter 13

Outlook

After presenting our résumé we take an outlook to directions for future research. Future work will concern the application of ProSet-Linda to more sophisticated problems than it was possible within the scope of this thesis, the evaluation of extensions and modifications to the proposed language features, and the implementation of the programming system.

13.1 Extending Matching to Unification

Proposals for combining generative communication with PROLOG often replace Linda's matching by PROLOG's unification [Sutcliffe and Pinakis, 1990; Anderson *et al.*, 1990; Bosschere and Wulteputte, 1991; Sutcliffe, 1993; Bosschere *et al.*, 1993]. To some extent, PROLOG's database of rules can be compared with Linda's tuple space and a specified goal can be compared with a template. A PROLOG interpreter tries to unify a goal with the database of rules, usually via backtracking. A tuple-space manager tries to match templates with tuples in tuple space.

However, the main difference between Linda's matching and PROLOG's unification is that unification reports failure when no answer substitution is possible, whereas matching blocks when no matching tuple is available. PROLOG's closed world assumption (one of PROLOG's fundamentals) implies that an execution simply *cannot* wait for missing information. Another difference is that with PROLOG's unification several substitutions may be returned. With Linda's matching at most one substitution (a tuple) may be returned.

In addition, backtracking might become a problem in tuple space. Adding backtrackable tuple-space operations to PROLOG implies backtrackable communications primitives. Backtracking of communications among distributed processes is very complex to implement and would require something like a temporal logic framework to define the semantics [Ciancarini, 1992].

Therefore, the extension of ProSet's conditional value matching to PROLOG-like unification in tuple space would cause serious problems for the definition of the semantics and for the implementation. However, in Linda/Q [Hiroyuki and Masaaki, 1991] nested tuples and templates are proposed to extend matching to unification. Linda/Q is proposed as an extension to C, and not as an extension to PROLOG. ProSet already supports nested tuples. Therefore, it seems to be promising to have nested templates, too. It would be straightforward to extend the matching rules for nested tuples and templates. The advantage would be increased flexibility for the programmer, but also increased complexity. The elaboration of such an extension to matching is beyond the scope of this thesis and, furthermore, we would like to gain more practical experience with the available features for matching to justify the need for more flexibility (and also more complexity), before extending the system.

13.2 An Implementation of a Graphical Debugger

The standard approach to debugging a sequential program is to run it and to stop the execution at some point in the program (usually called *breakpoint*). At such a breakpoint the program state is checked for any errors. If an error is discovered, the appropriate correction can be made in the original source program. If no errors are discovered, the program's execution is continued to some other breakpoint, or the program is re-started and stopped at an earlier breakpoint. Again the program state is checked. This method of error detection and correction is called *traditional cyclic debugging* [McDowell and Helmbold, 1989]. Another method of error detection is to monitor the execution of the program by placing `put` statements at useful points in the program which indicate the path of execution, and the position at which the program execution failed. This attempts to narrow down the area in which the error is likely to occur.

The above mentioned techniques are characteristic of many sequential debuggers/monitors, and are sufficient for sequential programs. However, debugging/monitoring parallel programs is more difficult [McDowell and Helmbold, 1989]:

- Nondeterminism arising from race conditions is normally beyond the user's control.
- Attempts to gain more information about the system by, for example, placing `put` statements in the program may alter a crucial race condition, and cause the erroneous behavior to disappear. This phenomenon is called the *probe effect*.
- The absence of a synchronized global clock makes it difficult the determine the precise order of events on distinct, parallel executing processors.

Therefore, the design of a debugger for a parallel language requires particular attention. So-called *event-based* approaches intend to solve the problems [McDowell and Helmbold, 1989]. The debugger for ProSet-Linda should support a single-step mode where a breakpoint is automatically inserted at each tuple-space operation (these are the *events* for our debugger). This event-based approach to monitoring and debugging provides a graphical presentation of a parallel program as a series of states. The debugger should also support to record an *event history* in a file containing all of the events generated by a program. The system can then *replay* the parallel program based on the contents in a history file. The replay of a program reduces the probe effect [LeBlanc and Mellor-Crummey, 1987].

It is straightforward for a graphical debugger for ProSet-Linda to present each tuple space in a separate window. This is somewhat similar to the way TupleScope [Bercovitz and Carriero, 1990] presents each partition of C-Linda's tuple space. The difference is that in ProSet the separation is defined by the programmer, whereas in C-Linda the separation is based on compile-time analysis (see Sect. 11.3.1).

The graphical capabilities of today's workstations make it possible to construct levels of abstraction in the representation of tuples. At the top level, tuples could be given a uniform representation in the form of an icon appearing in a tuple-space window. The user would usually not need more specific information: useful information is already given by the existence of a tuple in a specific tuple space. When additional information about a tuple is needed, the user would click on the tuple icon and descend to the next more specific level of representation: a window containing the detailed textual representation of the tuple would then pop up. This approach is also applied in TupleScope. The representation in form of icons helps in managing the potentially large quantity of tuple data.

Levels of abstraction are also useful for the representation of processes. A process could be represented a the top level by an icon. This icon could change as the process changes its state, for example, from being blocked to continuing execution. More detailed information regarding the state of a process, such as the actual line of source code, could be obtained by clicking on the process icon.

The ongoing work for implementing a graphical debugger for ProSet-Linda is the subject of Heiner Pohland's Master's thesis [Pohland, 1994].

13.3 An Implementation on a Local Area Network

To exploit the computational resources of workstations which are connected through a local area network, the implementation of the runtime system for ProSet-Linda on a local area network is underway. This is the subject of Ralf Naujokat's Master's thesis [Naujokat, 1994]. Multiple tuple spaces provide a direct approach for distributing the tuple space in a network. It is not necessary for the system to find application-specific mapping functions from tuples to processors, as proposed in [Wilson, 1991a]. Multiple tuple spaces simplify compile-time analysis with respect to partioning the tuple space and allow the programmer to partition the communication device as he sees fit.

At most sites, workstations are often idle. It would make sense if these idle workstations could help with ongoing parallel computations. A Linda-based model — called Piranha — has been proposed which incorporates these ideas [Carriero *et al.*, 1993]. The Piranha model is based on the master-worker model (see Sect. 6.1). In the Piranha model, a program is structured as a *cloud of tasks* which *computational Piranhas* attack. The more Piranhas attack, the faster the cloud is consumed and the faster the program completes. The role of Piranha is played by the workstations in the network when they become idle. *Idleness* is defined by the user. It depends on the keyboard idle time and the load average. A workstation leaves a computation when the user reclaims it by striking a key, moving the mouse, or doing something else with the workstation. The Piranha model demands a programming language which can accommodate dynamic process ensembles. The Piranha model is a form of *adaptive parallelism* [Carriero *et al.*, 1993]. Adaptive parallelism refers to parallel computations on a dynamically changing set of processors: processors may join or withdraw from the computation as it proceeds. The implementation on a local area network will be used to experiment with adaptive parallelism in ProSet-Linda.

13.4 Optimizing Analysis of Tuple-Space Access

[Jagannathan, 1991] analyses multiple tuple spaces in Scheme-Linda based on an inference engine that statically computes the set of tuples (and their structural attributes) which can occupy any given tuple space. The result of this inference procedure can be used to select appropriate data structures for the representation of individual tuple spaces. Such an analysis may also be useful in ProSet to select appropriate data structures for individual tuple spaces according to their use and content. Also, program transformations [Partsch, 1990] for tuple-space communication and optimal process creation may be useful.

Part V

Appendices

Appendix A

The Abstract Grammar for the Tuple-Space Operations

We present in this appendix a concise overview of the abstract grammar for the tuple-space operations in ProSet (see Sect. 5.4) using BNF (Backus Naur Form). Conversely, the informal semantics in Chap. 5 is presented together with syntax diagrams, which are spread over the text. For a concise overview, we regard BNF as more appropriate. The abstract grammar:

```
Statement  ::= deposit Expr at Expr [ blockiffull ] end deposit ;
             | fetch FetchTemp at Expr [ else StmtList ] end fetch ;
             | meet MeetTemp at Expr [ else StmtList ] end meet ;
FetchTemp  ::= FetchTemp [ xor FetchTemp ]
             | ( [ FetchComp ] ) [ => StmtList ]
FetchComp  ::= FetchComp [ , FetchComp ]
             | Expr
             | ? [ LValue ] [ | Expr ]
MeetTemp   ::= MeetTemp [ xor MeetTemp ]
             | ( [ MeetComp ] ) [ => StmtList ]
MeetComp   ::= MeetComp [ , MeetComp ]
             | Expr
             | ? [ LValue ] [ | Expr ] [ into Expr ]
ExprList   ::= Expr [ , ExprList ]
StmtList   ::= Statement [ StmtList ]
```

Optional parts are enclosed in *[* and *]*. Terminal symbols are displayed in `typewriter` font. Note that the terminal symbol `|` is different from |, which denotes alternatives in the grammar. *FetchList* is similar to *MeetList*, except that `into`s are not allowed. *Statement* and *Expr* are not specified for the entire language. We refer to [Doberkat *et al.*, 1992a] for a definition of statements and expressions in ProSet.

Appendix B

The Specification Language Object-Z

This appendix provides a brief informal introduction to the formal specification language Object-Z. For a full account to plain Z we refer to [Diller, 1990]. [Spivey, 1992b] is the de-facto standard for plain Z and [Duke *et al.*, 1991] is the reference for Object-Z. The present appendix and the notes, which are enclosed in the symbols $\boxed{Z}$ and $\boxed{Z}$, are derived from these texts. We use the name Z when both languages, plain Z and Object-Z, are concerned.

An Object-Z specification document consists of interleaved passages of formal, mathematical text and informal prose explanation. The formal text consists of a sequence of paragraphs which gradually introduces the classes, schemas, global variables and basic types of the specification. Each paragraph builds on the ones which come before it (definition before use).

Types in Z are sets: every mathematical expression which appears in a Z specification is given a *type* determining a set known to contain the value of the expression. Each variable is given a type by its declaration. The *basic types* or *given sets* of a specification have no internal structure of interest. A given set may serve to the purpose of abstraction or generality. An object of the real world that does not need to be given a model at a particular abstraction level can be represented by a given set. The predefined basic types are $\mathbb{N}$ and $\mathbb{Z}$. $\mathbb{N}$ is the set of natural numbers $\{0, 1, 2, ...\}$, and $\mathbb{Z}$ is the set of integers. Basic type definitions introduce new basic type names, which become part of the global vocabulary of basic types. They are introduced as follows:

$$[PERSON, IDENTITY]$$

Such a basic type definition introduces one or more basic types. These names must not have a previous global declaration, and their scope extends from the definition to the end of the specification. From these *atomic objects*, composite objects can be put together in various ways. These composite objects are the members of composite types put together with the type constructors of Z. There are three kinds of composite types: set types, Cartesian product types (tuples), and schema types.

An *abbreviation definition* introduces a new global constant, which may later be used as an abbreviation for the specified expression. The following example introduces the name *BIJECTION* as an abbreviation for the set of bijections from *PERSON* to *IDENTITY*:

$$BIJECTION == PERSON \rightarrowtail\!\!\!\!\twoheadrightarrow IDENTITY$$

All kinds of functions, such as bijections, are relations with appropriate implicit constraints. Relations are sets of pairs. Pairs are tuples with two components.

The formal part of an Object-Z specification makes use of two-dimensional graphical constructs for classes, schemas, axiomatic descriptions, and generic descriptions. Sections B.1 to B.5 introduce the concepts of plain Z and Sect. B.6 introduces the object-oriented extensions of Object-Z. We conclude this appendix with remarks concerning the usability of Z and Object-Z in Sect. B.7.

B.1 Schemas

Schemas provide a means for structuring specifications in plain Z. They can be used to describe both the static aspect of a system (the state space and invariant relations on the state), and the dynamic aspects (the operations which change the state). The general form of schemas in Z is as follows:

```
┌─ S ──────────────────────────────
│ D
├──────────
│ P
└──────────────────────────────────
```

where S is the name of the schema, D is a declaration and P a predicate. D is also called the *signature* of S. For instance the declaration $x, y : \mathbb{Z}$ in a schema introduces the variables x and y with type $\mathbb{Z}$, which are then called the *components* of this schema. Such variables are local to the respective schema, unless the schema is included elsewhere. A schema is included through using its name as a declaration. The component names are then visible within the scope of the respective declaration. P is also called the *property* of the schema. When multiple schemas are combined, then their signatures are joined, and their properties are combined accordingly. The schema name S is global.

There are some standard decorations for names used in describing operations: $'$ for labeling the final state of an operation, ? for labeling its inputs, and ! for labeling its outputs. If we decorate a schema name, this means a copy of this schema in which all the component names have been decorated accordingly.

It is possible to have generic schemas:

```
┌─ S[X₁, ..., Xₙ] ─────────────────
│ D
├──────────
│ P
└──────────────────────────────────
```

where the X_i are the formal generic parameters which can occur in the types assigned to the identifiers in the declaration D. Later when the generic object is used, *actual generic parameters* (set-valued) are supplied. These determine the sets which the formal parameters take as their values. The above generic schema might be instantiated as follows:

$$I \mathrel{\widehat{=}} S[A_1, \ldots, A_n]$$

where the A_i are the actual generic parameters (set-valued) and I is the instantiated schema. New schemas may also be defined this way (via $\widehat{=}$) by combining old ones with the operations of the *schema calculus* [Spivey, 1992b]. For example, the effect of the schema $\lor$ operator is to make a schema in which the predicate part is the result of joining the predicate parts of its arguments with the logical connective $\lor$.

The schema $\Delta State$ is implicitly defined as the combination of the *before*-state $State$ and the *after*-state $State'$ whenever a schema $State$ is introduced:

```
┌─ ΔState ─────────────────────────
│ State
│ State'
└──────────────────────────────────
```

This implicit definition may be overridden by explicit definitions. The schema $\Xi State$ is implicitly defined as the state space of a data type whenever a schema $State$ is introduced:

```
┌─ ΞState ─────────────────────────
│ State
│ State'
├──────────
│ θState' = θState
└──────────────────────────────────
```

Such implicit definitions may be overridden by explicit definitions. $\theta State$ is the binding formation of values to the components of the schema *State*. Let the components of *State* be $x_1, ..., x_n$. The following law holds:

$$\theta State' = \theta State \Leftrightarrow x_1' = x_1 \wedge ... \wedge x_n' = x_n$$

Note that we do *not* use the Ξ-construction in our specification and that we use Δ-lists (Sect. B.6) instead of the above described Δ-construction.

A state schema groups together variables and defines the relationship that holds between their values. At any instant, these variables define the state of the system which they model. An operation schema defines the relationship between the *before* and *after* states corresponding to one or more state schemas.

B.2 Axiomatic Descriptions

The general form of an axiomatic description is as follows:

```
│ D
├──────────
│ P
```

where D is a declaration which introduces one or more global variables and P is an optional predicate that constrains the values that can be taken by the variables introduced in D. The variables declared in D cannot have been previously declared globally and their scope extends to the end of the specification. D becomes part of the *global signature* of the specification and P contributes to a *global property*.

A predicate may appear on its own as a paragraph; it specifies a constraint on the values of previously declared global variables. The effect is as if the constraint had been stated as part of the axiomatic descriptions in which the variables were introduced. Therefore, the predicate P in the above axiomatic description box could also be placed behind this box without changing the semantics.

B.3 Generic Descriptions

The general form of a generic description is as follows:

```
╒═[X1, ..., Xn]══════════════════════
│ D
├──────────
│ P
└────────────────────────────────────
```

where the X_i are the formal generic parameters which can occur in the types assigned to the identifiers in the declaration D. The variables declared in D cannot have been previously declared globally and their scope extends to the end of the specification. The predicate P constrains the identifiers introduced in D.

Generic descriptions may be used to define concepts such as relations, functions, sequences, bags and the operations on them.

B.4 Free Type Definitions

A new basic type may also be introduced by a *free type definition*, where recursive structures are allowed. A free type definition such as

$$T ::= Constant \mid PtoT\langle\langle PERSON\rangle\rangle \mid next\langle\langle T\rangle\rangle$$

is equivalent to the basic type definition

$$[T]$$

extended by the axiomatic description

$$\begin{array}{|l}
Constant : T \\
PtoT : PERSON \rightarrowtail T \\
next : T \rightarrowtail T \\
\hline
\langle \{Constant\}, \operatorname{ran} PtoT, \operatorname{ran} next \rangle \ \mathsf{partition}\ T
\end{array}$$

$X \rightarrowtail Y$ is the set of total injections from X to Y. Finite sequences are enclosed in $\langle$ and $\rangle$. Sets are enclosed in $\{$ and $\}$. ran yields the range of a relation. The left-hand operand of partition has to be an indexed family of sets. partition holds when all these sets are disjoint and when their union is equal to the right-hand operand. ran and partition are keywords in Z. A particular common example of an indexed family of sets is a sequence of sets, which is at base only a function defined on a subset of $\mathbb{N}$.

Note that the device of defining free type definitions adds nothing to the power of Z. It is a convenient shorthand.

B.5 Expressions

The expressions within the boxes and in global constraints are based on first-order logic and set theory. Sets, tuples (with at least two components), and bindings of values to components of schemas are fundamental to Z. Relations, functions, bags (multisets), and finite sequences belong to the basic mathematical tool-kit of Z. Sequences may be empty. Here, we only present an example for a set-forming expression in Z:

$$\{ x : \mathbb{N} \mid x \leq 10 \bullet x * x \}$$

which corresponds to

```
{ x * x:  x in ℕ | x <= 10 }
```

in ProSet. However, this is not a legal expression in ProSet, since $\mathbb{N}$ is an infinite set. Note that compound objects in Z have to be homogeneous, whereas compound objects in ProSet may be heterogeneously composed. The type of the above Z expression is "set of natural numbers" and the type of the above ProSet expression is simply "set". The bound variables are local to the respective constructs in both languages.

B.6 Classes

Inferring the operation schemas that may affect a particular state schema in plain Z requires examining the signatures of all operation schemas. In large specifications this is impracticable. Object-Z overcomes this problem by introducing classes. State variables and related operations are encapsulated into classes. Inheritance facilitates the construction of complex specifications by allowing components to include the states and operations from simpler components. A simple example class specification:

MyClass

$\upharpoonright(x, \mathit{Increment}, \mathit{Decrement})$ [visibility list]

$\mathit{ParentClass}$ [**remove** $\mathit{SomeFeature}$] [inherited classes]

$x, y : \mathbb{Z}$ [variables]

$x = y$ [class invariant]

Init

$x = y = 1$ [initial state]

Increment

$\Delta(x, y)$

$y' = x' = x + 1$

Decrement

$\Delta(x, y)$

$y' = x' = x - 1$

$\Box(\mathbf{op} = \mathit{Increment} \Rightarrow \Diamond(\mathbf{op} = \mathit{Decrement}))$ [history invariant]

We could *instantiate* objects of this class via declarations like "*MyObject* : *MyClass*". A class is a template for objects of that class: for each such object, its states are instances of the class' state schema and its individual state transitions conform to individual operations of the class. An Object is said to be an instance of a class and to evolve according to the definitions of its class.

The [visibility list] gives those features externally visible to the clients of an object (users via instantiation). If none is given, the default is that all operations are visible. The symbol $\upharpoonright$ is used to indicate visibility. In our example, the state variable x, and the operations *Increment* and *Decrement* are externally visible to clients, whereas the state variable y is hidden.

The attributes (constants and state variables) and the operations are collectively called the *features* of a class. The [inherited classes] are the names of super-classes to be inherited; a subclass incorporates all the features of its super-classes, including their operations and invariants. Visibility is not inherited so that a derived class may nominate any inherited feature as visible. It is not possible in Object-Z to indicate which features of an object are available to its children (users through inheritance). Therefore, children always have access to all the features of their parents. However, children may optionally restrict their access via the keyword **remove** while inheriting a class, as it has been done in the example with *SomeFeature*. Under multiple inheritance, features having the same name are merged (semantic identification).

The values of the [variables] are constrained by the [class invariant]. The initial state schema is distinguished by having as its name the keyword *Init*. The predicate $a = b = c$ is an abbreviation for $a = b \wedge b = c$.

The operations are defined using schemas, in a way very similar to plain Z, defining a relation between before and after state. An operation's Δ-list contains a subset of the variables which are declared, either implicitly or explicitly, in both unprimed and primed form in the operations signature. The understanding is that when the operation is applied to an object of the class, those variables not in the list are unchanged. By convention, not indicating a Δ-list is equivalent to specifying an empty Δ-list.

The set of all possible histories is restricted by history invariants. Such history invariants are liveness and fairness properties, which explicitly restrict the set of allowable histories by means of temporal logic. The set of possible histories of a class is initially determined by the class state (including the initial state) and the allowable operations, and can then be further restricted by incorporating history invariants. Typically, a history invariant will be concerned with specifying liveness (progress) issues, for example, when specifying fairness of operations, or the order in which operations may or may not occur.

In history invariants, the keyword **op** denotes the name of an operation in the history. The temporal logic notation can be used within history invariants:

$\Box p$	Predicate p holds at every future state in the history.
$\Diamond p$	Eventually there is a future state at which predicate p holds.

For example, $p \Rightarrow \Diamond q$ says that if p holds in the current state, q will eventually hold in a future state. The history invariant in our example means that *always* after an *Increment* operation, *eventually* a *Decrement* operation must occur. We refer to [Emerson, 1990] for a full account to temporal logic.

B.7 Remarks Concerning the Usability of Z and Object-Z

While constructing the presented formal specification (Chaps. 8 and 9) we recognized an important drawback of using Object-Z or Z with the *f*UZZ package [Spivey, 1992a]: the principle of *definition before use*, which leads to a bottom-up development of the specification. It would be more natural to write and explain the specification in a top-down manner. The important point with Z is just that any specification must be written in a way such that its definitions can be ordered to satisfy the principle of definition before use [Spivey, 1992b, page 47]. This avoids recursive definitions in which a schema includes itself. Therefore, it should be possible to develop a tool for Z that allows the introduction of paragraphs in any order and ensures that the principle of definition before use can be satisfied. We propose a *f*UZZ directive, which *announces* a forthcoming definition. The existing *f*UZZ directives allow preliminary, invisible definitions, but the later final definitions cannot be type-checked because they are redefinitions of global names. One could copy the formulae of the final definition into the preliminary, invisible definition, but then it would become impractical to change the specification.

Additionally, it is not possible in Object-Z to indicate which features of a class are available to its children (users through inheritance). Therefore, children always have access to all the features of their parents. However, children may restrict their access while inheriting a class. Our proposal for enhancing Object-Z is to split the definition of a class in private and public parts, as one can do in C++ [Stroustrup, 1986]. This way one could hide auxiliary definitions from children and also from clients. We think that children should not be responsible for restricting their access while inheriting a class, but clients should be able to restrict their access to an instantiated object. This philosophy seems to be somewhat the opposite to the principles applied in Object-Z.

Refinement is not sufficiently supported by Z: we had to explicitly rewrite the entire abstract operations in Chap. 9 to obtain the concrete operations despite the fact that we refined only parts of the schemas.

Appendix C

Types of All Names Defined Globally

This appendix has been produced by the *fuzz* type-checker for Z with the `-t` flag [Spivey, 1992a]. A comprehensive overview of all names defined globally in the specification with their associated types is given as *fuzz* sees it: the class structure is not visible in this list. Some line breaks were inserted to fit into two columns. An index to these global names may be found in the index of formal definitions at the end of this document.

```
Given Expression

Given LValue

Given Statement

Given Process

Var Execute _: P Statement

Given Value

Var atom: Value

Var boolean: Value

Var integer: Value

Var real: Value

Var string: Value

Var tuple: Value

Var set: Value

Var function: Value

Var modtype: Value

Var instance: Value

Var TRUE: Value

Var FALSE: Value

Var om: Value

Var ValuesOfType: Value -+> P Value

Var Type: Value -+> Value

Var Evaluate: Expression -+> Value

Var ProcRetVal: Process -+> Value

Var _ \IsAssigned _: LValue <-> Value

Given TupleComp

Var TupleValue: Value -+> TupleComp

Var TupleProcess: Process -+> TupleComp

Abbrev APTuple: P (seq TupleComp)

Genconst Arity[1]:
      (Value -+> @1) -+> (seq @1 -+> NN)

Var TupArity: APTuple -+> NN

Given OptLValue

Var NoLValue: OptLValue

Var IsLValue: LValue -+> OptLValue

Given OptInto

Var NoInto: OptInto

Var IsInto: Expression -+> OptInto

Schema Formal
    Destination: OptLValue
    Into: OptInto
End

Given TempComp

Var TempValue: Value -+> TempComp

Var TempFormal: Formal -+> TempComp
```

```
Schema Template
    List: seq TempComp
    Condition: Expression
End

Var TempArity: seq TempComp -+> NN

Var _ \FormalAssign _: Template <-> APTuple

Var _ \EvalIntos _: Template x APTuple -+> APTuple

Var _ \CompMatches _: TupleComp <-> TempComp

Var _ \Matches _: APTuple <-> Template

Given OptStmt

Var NoStmt: OptStmt

Var IsStmt: Statement -+> OptStmt

Var OptExecute _: P OptStmt

Given OpType

Var FetchOp: OpType

Var MeetOp: OpType

Var MeetIntoOp: OpType

Schema Pending
    temp: Template
    os: OptStmt
    proc: Process
    type: OpType
End

Schema TupleSpace
    Id: Value
    Limit: Value
    Tuples: bag APTuple
    PendTemp: bag Pending
    PendFull: Process -++> APTuple
End

Var _ \SAR _:
   F Process x bag Pending -+> bag Pending

Var IDsOF: F TupleSpace -+> F Value

Schema ProgramState
    TSs: F TupleSpace
    ActiveProcs: F Process
End

Schema \Delta ProgramState
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
End

Schema \Init
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
End

Schema ProgramTermination
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
End

Var ActuallyActiveProcesses: F Process

Var ActuallyPendingProcesses: F Process

Var ActuallyExistingProcesses: F Process

Schema ProcessCreation
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    NewProcess?: Process
End

Schema ProcessTermination
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    ToKill?: Process
End

Abbrev \TypeMismatch: Statement

Abbrev \InvalidId: Statement

Abbrev \ExcTSisFull: Statement

Schema CreateTSok
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InLimit?: Value
    Return!: Value
End

Schema \Xi ProgramState
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
End

Schema CreateTSTypeMismatch
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InLimit?: Value
    Exception!: Statement
End

Schema CreateTS
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InLimit?: Value
    Return!: Value
    Exception!: Statement
End

Schema ExistsTS
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTS?: Value
    Return!: Value
    Exception!: Statement
End

Schema ClearTSok
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTS?: Value
    Return!: Value
End

Schema ClearTSinvalid
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTS?: Value
    Exception!: Statement
End

Schema ClearTS
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTS?: Value
    Return!: Value
    Exception!: Statement
End

Schema RemoveTSfromState
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTS?: Value
End

Schema RemoveTS
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTS?: Value
    Return!: Value
    Exception!: Statement
End

Var HasIntos _: P Template

Genconst BagSum[1]: bag @1 -+> NN

Var IntValueOf: Value -+> ZZ

Given BlockMode

Var BlockIfFull: BlockMode

Var DoNotBlock: BlockMode

Abbrev TempList:
   P (seq (Template x OptStmt) x Value)

Var MakePends:
   TempList x Process x OpType -+> bag Pending

Genconst GetTemp[3]: @1 x @2 x @3 -+> @1

Genconst GetTup[3]: @1 x @2 x @3 -+> @2

Genconst GetOS[3]: @1 x @2 x @3 -+> @3

Var _ \AddTuple _:
 (F TupleSpace x F Process) x (APTuple x Value)
        -+> F TupleSpace x F Process

Schema DepositOK
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    ToDeposit?: APTuple x Value
End

Schema DepositInvalid
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    ToDeposit?: APTuple x Value
    Exception!: Statement
End

Schema TSisFull
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    ToDeposit?: APTuple x Value
End

Schema FullTSBlock
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    ToDeposit?: APTuple x Value
    Blocking?: BlockMode
    InProc?: Process
End

Schema FullTSException
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    ToDeposit?: APTuple x Value
    Blocking?: BlockMode
    Exception!: Statement
End

Schema Deposit
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    ToDeposit?: APTuple x Value
    Blocking?: BlockMode
    InProc?: Process
    Exception!: Statement
End

Schema DoElseStmt
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InProc?: Process
    InTempList?: TempList
    Else?: OptStmt
End

Schema InvalidTempList
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InProc?: Process
    InTempList?: TempList
    Exception!: Statement
End

Schema FetchMatch
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InProc?: Process
    InTempList?: TempList
End

Schema DisallowIntos
    InTempList?: TempList
End

Schema FetchNoMatch
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InProc?: Process
    InTempList?: TempList
    Else?: OptStmt
End

Schema Fetch
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InProc?: Process
    InTempList?: TempList
    Else?: OptStmt
    Exception!: Statement
End

Schema MeetMatch
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InProc?: Process
    InTempList?: TempList
End

Schema MeetNoMatch
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InProc?: Process
    InTempList?: TempList
    Else?: OptStmt
End

Schema Meet
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InProc?: Process
    InTempList?: TempList
    Else?: OptStmt
    Exception!: Statement
End

Schema NoOp
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
End

Schema TupleSpaceD
    IdD: Value
    LimitD: Value
    TuplesD: bag APTuple
    PendTempD: seq Pending
    PendFullD: seq (Process x APTuple)
End

Schema TSAbstraction
    Id: Value
    Limit: Value
    Tuples: bag APTuple
    PendTemp: bag Pending
    PendFull: Process -++> APTuple
    IdD: Value
    LimitD: Value
    TuplesD: bag APTuple
    PendTempD: seq Pending
    PendFullD: seq (Process x APTuple)
End

Schema ProgramStateD
    TSsD: F TupleSpaceD
    ActiveProcsD: F Process
End

Schema ProgramAbstraction
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSsD: F TupleSpaceD
    ActiveProcsD: F Process
End

Var _ \SARSEQ _:
   F Process x seq Pending -+> seq Pending

Var _ \AddTupleD _:
 (F TupleSpaceD x F Process) x (APTuple x Value)
        -+> F TupleSpaceD x F Process

Var IDsOFD: F TupleSpaceD -+> F Value

Var MakePendsD:
   TempList x Process x OpType -+> seq Pending

Schema \Delta ProgramStateD
    TSsD: F TupleSpaceD
    ActiveProcsD: F Process
    TSsD': F TupleSpaceD
    ActiveProcsD': F Process
End

Schema DepositOKD
    TSsD: F TupleSpaceD
    ActiveProcsD: F Process
    TSsD': F TupleSpaceD
    ActiveProcsD': F Process
    ToDeposit?: APTuple x Value
End

Schema FullTSBlockD
    TSsD: F TupleSpaceD
    ActiveProcsD: F Process
    TSsD': F TupleSpaceD
    ActiveProcsD': F Process
    ToDeposit?: APTuple x Value
    Blocking?: BlockMode
    InProc?: Process
End

Schema FetchMatchD
    TSsD: F TupleSpaceD
    ActiveProcsD: F Process
    TSsD': F TupleSpaceD
    ActiveProcsD': F Process
    InProc?: Process
    InTempList?: TempList
End

Schema FetchNoMatchD
    TSsD: F TupleSpaceD
    ActiveProcsD: F Process
    TSsD': F TupleSpaceD
    ActiveProcsD': F Process
    InProc?: Process
    InTempList?: TempList
    Else?: OptStmt
End
```

Indices

Index of Formal Definitions

Index of Explained Object-Z Symbols and Keywords

General Index

Bibliography

[Abarbanel, 1991] R. Abarbanel. Distributed object mangement with Linda. Technical report, Boeing Computer Services, Seattle, WA, August 1991. (cited on p. 34, 75, 76, 80, 82, 83)

[Abelson *et al.*, 1987] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and interpretation of computer programs.* MIT Press, 1987. (cited on p. 51)

[Ackermann, 1982] W.B. Ackermann. Data flow languages. *IEEE Computer*, 15(2):15–25, 1982. (cited on p. 15)

[Agha and Callsen, 1993] G. Agha and C.J. Callsen. ActorSpace: an open distributed programming paradigm. In *Proc. Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 23–32, San Diego, CA, May 1993. (cited on p. 34)

[Agha, 1986] C. Agha. *Actors: A model of concurrent computation in distributed systems.* The MIT Press, 1986. (cited on p. 22)

[Ahmed and Gelernter, 1992] S. Ahmed and D. Gelernter. A CASE Environment for Parallel Programming. In *Proc. Fifth International Workshop on Computer-Aided Software Engineering.* IEEE Computer Society Press, July 1992. (cited on p. 34, 76)

[Aho *et al.*, 1986] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers — Principles, Techniques, and Tools.* Addison Wesley, 1986. (cited on p. 44)

[Ahuja *et al.*, 1986] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, 1986. (cited on p. 33, 65)

[Ahuja *et al.*, 1988] S. Ahuja, N. Carriero, D. Gelernter, and V. Krishnaswamy. Matching language and hardware for parallel computation in the Linda machine. *IEEE Transactions on Computers*, 37(8):921–929, August 1988. (cited on p. 144, 145)

[Anderson and Shasha, 1992] B.G. Anderson and D. Shasha. Persistent Linda: Linda + Transactions + Query Processing. In Banâtre and Métayer [1992], pages 93–109. (cited on p. 75, 76, 79, 82–84)

[Anderson *et al.*, 1990] H. Anderson, P.D. Fabricius, and M.G. Jensen. Linda & Logic. Master's thesis, University of Aalborg, Denmark, June 1990. (cited on p. 5, 25, 34, 151)

[Anderson, 1991] B.G. Anderson. *Persistent Linda – Adding Transactions to a Parallel Programming Model.* PhD thesis, Courant Institute, New York University, NY, April 1991. (cited on p. 81)

[Andrews and Schneider, 1983] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, 1983. (cited on p. 18, 21)

[Andrews, 1991] G.R. Andrews. *Concurrent Programming.* Benjamin/Cummings, 1991. (cited on p. 18, 57, 61, 70, 72)

[Arango and Berndt, 1989] M. Arango and D. Berndt. TSnet: A Linda implementation for networks of Unix-based computers. Research Report 739, Yale University, New Haven, CT, August 1989. (cited on p. 142)

[Arvind *et al.*, 1989] Arvind, R.S. Nikhil, and K.K. Pingali. I-Structures: data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, 1989. (cited on p. 52)

[Bakken and Schlichting, 1993] D.E. Bakken and R.D. Schlichting. Supporting fault-tolerant parallel programming in Linda. Technical Report TR93-18, The University of Arizona, Tucson, AZ, June 1993. (cited on p. 83)

[Bal and Kaashoek, 1993] H.E. Bal and M.F. Kaashoek. Object distribution in Orca using compile-time and run-time techniques. In *Proc. Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '93)*, Washington, DC, September 1993. (cited on p. 31, 146)

[Bal *et al.*, 1989] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, 1989. (cited on p. 17, 18)

[Bal *et al.*, 1992] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992. (cited on p. 29–31, 146)

[Bal, 1990] H.E. Bal. *Programming Distributed Systems.* Silcon Press, 1990. (cited on p. 31, 35)

[Bal, 1992] H.E. Bal. A comparative study of five parallel programming languages. *Future Generations Computer Systems*, 8:121–135, 1992. (cited on p. 60, 74, 79, 146)

[Banâtre and Métayer, 1990] J.P. Banâtre and D. Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15(1):55–77, 1990. (cited on p. 27)

[Banâtre and Métayer, 1992] J.P. Banâtre and D. Le Métayer, editors. *Research Directions in High-Level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science.* Springer-Verlag, 1992. (cited on p. 181, 191, 195)

[Banâtre and Métayer, 1993] J.P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, 1993. (cited on p. 16, 27, 38)

[Banerjee *et al.*, 1993] U. Banerjee, R. Eigenmann, A. Nicolau, and D.A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993. (cited on p. 15)

[Barendregt, 1985] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1985. (cited on p. 23)

[Barnes, 1984] J.G.P. Barnes. *Programming in Ada.* Addison-Wesley, 2nd edition, 1984. (cited on p. 75)

[Ben-Ari, 1990] M. Ben-Ari. *Principles of concurrent and distributed programming.* Prentice-Hall, 1990. (cited on p. 16, 61, 70, 72)

[Bercovitz and Carriero, 1990] P. Bercovitz and N. Carriero. TupleScope: A graphical monitor and debugger for Linda-based parallel programs. Research Report 782, Yale University, New Haven, CT, April 1990. (cited on p. 34, 152)

[Berry and Boudol, 1990] G. Berry and G. Boudol. The chemical abstract machine. In *Proc. Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 81–94, San Francisco, CA, January 1990. (cited on p. 28)

[Bettermann, 1992] S. Bettermann. The Implementation of Linda in C and Associated Problems. Research Report 92/2, Edith Cowan University, Mount Lawley WA, Australia, 1992. (cited on p. 142)

[Bjornson *et al.*, 1988] R. Bjornson, N. Carriero, D. Gelernter, and J. Leichter. Linda, the portable parallel. Research Report 520, Yale University, New Haven, CT, 1988. (cited on p. 33)

[Bjornson *et al.*, 1991] R. Bjornson, N. Carriero, D. Gelernter, T. Mattson, D. Kaminsky, and A. Sherman. Experience with Linda. Research Report 866, Yale University, New Haven, CT, August 1991. (cited on p. 33, 34)

[Bjornson, 1992] R.D. Bjornson. *Linda on Distributed Memory Multiprocessors.* PhD thesis, Yale University, New Haven, CT, November 1992. (cited on p. 78, 142, 143)

[Blum, 1982] B.I. Blum. The life cycle — a debate over alternate models. *ACM SIGSOFT Software Engineering Notes*, 7(4):18–20, 1982. (cited on p. 6, 12)

[Boehm, 1976] B.W. Boehm. Software engineering. *IEEE Transactions on Computers*, 25(12):1226–1241, 1976. (cited on p. 6)

[Boehm, 1981] B.W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981. (cited on p. 3)

[Bogoch *et al.*, 1990] S. Bogoch, I. Bason, J. Williams, and M. Russel. Supercomputers get personal. *BYTE*, 15(5):231–237, 1990. (cited on p. 142)

[Borrman *et al.*, 1989] L. Borrman, M. Herdieckerhoff, and A. Klein. Tuple space integrated into Modula-2, Implementation of the Linda concept on a hierarchical multiprocessor. In Jesshope and Reinartz, editors, *Proc. CONPAR'88*, New York, 1989. Cambridge Univ. Press. (cited on p. 34, 141)

[Bosschere and Wulteputte, 1991] K. De Bosschere and L. Wulteputte. Multi-Prolog: Implementation on an 88000 Shared Memory Multiprocessor. Technical Report DG 91-19, University of Gent, LEM, Gent, Belgium, December 1991. (cited on p. 5, 34, 151)

[Bosschere *et al.*, 1993] K. De Bosschere, J.-M. Jacquet, and P. Tarau, editors. *Proc. ICLP'93 Post-Conference Workshop on Blackboard-Based Logic Languages*, Budapest, Hungary, June 1993. (cited on p. 34, 151)

[Bréant and Pavoit-Adet, 1992] F. Bréant and E. Pavoit-Adet. Occam prototyping from hierarchical Petri nets. Technical Report MASI 92.08, University of Paris 6, Institut Blaise Pascal, Paris, France, February 1992. (cited on p. 18)

[Broadbery and Playford, 1991] P. Broadbery and K. Playford. Using object-oriented mechanisms to describe Linda. In Wilson [1991b], pages 14–26. (cited on p. 34, 75, 76)

[Budde *et al.*, 1984] R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllighoven, editors. *Approaches to Prototyping.* Springer-Verlag, 1984. (cited on p. 11)

[Budde *et al.*, 1992] R. Budde, K. Kautz, K. Kuhlenkamp, and H. Züllighoven. *Prototyping — An Approach to Evolutionary System Development.* Springer-Verlag, 1992. (cited on p. 6, 11, 13, 14)

[Butcher and Zedan, 1991] P. Butcher and H. Zedan. Lucinda — an overview. *SIGPLAN Notices*, 26(8):90–100, 1991. (cited on p. 34)

[Butcher, 1991] P. Butcher. A behavioural semantics for Linda-2. *Software Engineering Journal*, 6(4):196–204, July 1991. (cited on p. 5)

[Callsen *et al.*, 1991] C.J. Callsen, I. Cheng, and P.L. Hagen. Optimizing Linda. Master's thesis, University of Aalborg, Denmark, June 1991. (cited on p. 4, 34, 142)

[Cannon, 1992] S.R. Cannon. Experience with a tuple-space approach for parallel compilation of LR languages. In *Proc. 21st International Conference on Parallel Processing*, pages II–218–II–225, St. Charles, IL, August 1992. (cited on p. 34)

[Carriero and Gelernter, 1986] N. Carriero and D. Gelernter. The S/Net's Linda kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, 1986. (cited on p. 141, 143)

[Carriero and Gelernter, 1988] N. Carriero and D. Gelernter. Applications experience with Linda. In *Proc. ACM Symposium on Parallel Programming*, pages 173–187, New Haven, CT, July 1988. (cited on p. 34)

[Carriero and Gelernter, 1989] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989. (cited on p. 33)

[Carriero and Gelernter, 1990a] N. Carriero and D. Gelernter. *How to write parallel programs.* MIT Press, 1990. (cited on p. 32, 33, 70, 71, 74, 76)

[Carriero and Gelernter, 1990b] N. Carriero and D. Gelernter. Tuple analysis and partial evaluation strategies in the Linda precompiler. In Gelernter et al. [1990]. (cited on p. 144)

[Carriero and Gelernter, 1992] N. Carriero and D. Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, February 1992. (cited on p. 2, 31, 33)

[Carriero *et al.*, 1986] N. Carriero, D. Gelernter, and J. Leichter. Distributed data structures in Linda. In *Proc. ACM Symposium on Principles of Programming Languages*, St. Petersburg, January 1986. (cited on p. 33)

[Carriero *et al.*, 1993] N. Carriero, D. Gelernter, D. Kaminsky, and J. Westbrook. Adaptive parallelism with Piranha. Research Report 954, Yale University, New Haven, CT, February 1993. (cited on p. 34, 153)

[Carriero, 1987] N. Carriero. *Implementation of tuple space machines.* PhD thesis, Yale University, New Haven, CT, December 1987. (cited on p. 141, 143)

[Chandra *et al.*, 1990] R. Chandra, A. Gupta, and J.L. Hennessy. Cool: a language for parallel programming. In Gelernter et al. [1990]. (cited on p. 23)

[Chandy and Misra, 1984] K.M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, 1984. (cited on p. 71)

[Chandy and Misra, 1988] K.M. Chandy and J. Misra. *Parallel Program Design.* Addison Wesley, 1988. (cited on p. 3, 25, 26)

[Chandy and Taylor, 1992] K.M. Chandy and S. Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett Publishers, 1992. (cited on p. 2, 26)

[Char, 1990] B.W. Char. Progress report on a system for general-purpose parallel symbolic algebraic computation. In *Proc. ISSAC '90*, pages 96–103, Tokyo, Japan, August 1990. (cited on p. 34, 141)

[Chiba *et al.*, 1991] S. Chiba, K. Kato, and T. Masuda. Optimization of distributed communication in multiprotocol tuple space. In *Proc. Third IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, December 1991. (cited on p. 142, 143)

[Christiansen *et al.*, 1987] M.G. Christiansen, M.M. Tanik, and S.L. Stepoway. Objective Linda: An object-centered perspective of Linda concepts, and issues of implementation. Technical Report 87-CSE-13, Southern Methodist University, Dallas, TX, June 1987. (cited on p. 34)

[Ciancarini *et al.*, 1992] P. Ciancarini, K.K. Jensen, and D. Yanklevich. The semantics of a parallel language based on a shared dataspace. Technical Report 26/92, University of Pisa, Pisa, Italy, July 1992. (cited on p. 4, 5, 112)

[Ciancarini, 1991] P. Ciancarini. PoliS: a programming model for multiple tuple spaces. In *Proc. Sixth International Workshop on Software Specification and Design*, pages 44–51, Como, Italy, October 1991. (cited on p. 34, 71, 80, 81)

[Ciancarini, 1992] P. Ciancarini. Parallel programming with logic languages: a survey. *Computer Languages*, 17(4):213–240, 1992. (cited on p. 25, 37, 151)

[CIP, 1985] Language Group CIP. *The Munich Project CIP. Volume 1: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985. (cited on p. 150)

[CIP, 1987] System Group CIP. *The Munich Project CIP. Volume 2: The Programm Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987. (cited on p. 14)

[Clayton *et al.*, 1990] P.G. Clayton, E.P. Wentworth, G.C. Wells, and F.K. de Heer-Menlah. An implementation of Linda Tuple Space under the Helios operating system. Technical Document PPG 90/8, Rhodes University, Grahamstown, South Africa, October 1990. (cited on p. 142, 145)

[Clocksin and Mellish, 1987] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 3rd edition, 1987. (cited on p. 25)

[Cocke, 1988] J. Cocke. The search for performance in scientific processors. *Communications of the ACM*, 31(3):249–253, 1988. (cited on p. 1)

[Cohen and Molinari, 1991] R. Cohen and B. Molinari. Implementation of C-Linda for the AP1000. In *Proc. Second Fujitsu-ANU CAP Workshop*, Camberra, Australia, November 1991. (cited on p. 141, 144)

[Craigen *et al.*, 1993] D. Craigen, S. Gerhart, and T. Ralston. Formal methods reality check: Industrial usage. In J.C.P. Woodcock and P.G. Larsen, editors, *Proc. FME '93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 250–267, Odense, Denmark, April 1993. Springer-Verlag. (cited on p. 4)

[Cunha *et al.*, 1989] J.C. Cunha, M.C. Ferreira, and L. Moniz Pereira. Programming in Delta Prolog. In G. Levi and M. Martelli, editors, *Proc. Sixth International Conference on Logic Programming*, pages 487–502, Lisbon, Portugal, 1989. MIT Press. (cited on p. 24, 37)

[Dahlen and MacDonald, 1990] U. Dahlen and N. MacDonald. Scheme-Linda. In Gupta [1990]. (cited on p. 34)

[Dearnley and Mayhew, 1983] P.A. Dearnley and P.J. Mayhew. In favour of system prototypes and their integration into the system development cycle. *The Computer Journal*, 26(1):36–42, 1983. (cited on p. 11, 13)

[DeMarco, 1978] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, 1978. (cited on p. 18)

[DeMillo *et al.*, 1979] R.A. DeMillo, R.J. Lipton, and A.J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979. (cited on p. 129)

[Dijkstra, 1971] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971. (cited on p. 70, 71)

[Dijkstra, 1975] W. Dijkstra. Guarded commands, nondetermininacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. (cited on p. 26)

[Dijkstra, 1988] E.W. Dijkstra. Position paper on "fairness". *ACM SIGSOFT Software Engineering Notes*, 13(2):18–20, April 1988. (cited on p. 78)

[Diller, 1990] A. Diller. *Z: An introduction to formal methods*. Wiley, 1990. (cited on p. 117, 126, 127, 131, 140, 150, 159)

[Doberkat and Fox, 1989] E.-E. Doberkat and D. Fox. *Software Prototyping mit SETL*. Leitfäden und Monographien der Informatik. Teubner-Verlag, 1989. (cited on p. 6, 11, 13, 14, 43)

[Doberkat *et al.*, 1990a] E.-E. Doberkat, U. Gutenbeil, and W. Hasselbring. SETL/E Sprachbeschreibung Version 0.1. Informatik-Bericht 01-90, University of Essen, March 1990. (cited on p. 43)

[Doberkat *et al.*, 1990b] E.-E. Doberkat, U. Gutenbeil, and W. Hasselbring. SETL/E — A Prototyping System based on Sets. In W. Zorn, editor, *Proc. TOOL'90*, pages 109–118. University of Karlsruhe, November 1990. (cited on p. 43)

[Doberkat *et al.*, 1992a] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, and C. Pahl. ProSet — Prototyping with Sets: Language Definition. Informatik-Bericht 02-92, University of Essen, April 1992. (cited on p. 43, 47, 58, 85, 87, 157)

[Doberkat *et al.*, 1992b] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, and C. Pahl. ProSet — A Language for Prototyping with Sets. In N. Kanopoulos, editor, *Proc. Third International Workshop on Rapid System Prototyping*, pages 235–248, Research Triangle Park, NC, June 1992. IEEE Computer Society Press. (cited on p. 43)

[Doberkat *et al.*, 1992c] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, and C. Pahl. A First Implementation of ProSet. In U. Kastens and P. Pfahler, editors, *International Workshop on Compiler Construction CC'92 (Poster Session)*, pages 23–27. University of Paderborn, Informatik-Bericht Nr. 103, October 1992. (cited on p. 43, 131, 140)

[Doberkat *et al.*, 1993] E.-E. Doberkat, W. Franke, and W. Hasselbring. Software Prototyping — Persistenz und Parallelität. In M. Nagl, editor, *Proc. 16th European Congress Fair for Technical Communications (ONLINE'93), Congress VI (Software and Information Engineering)*, pages C622.01–C622.20, Hamburg, Germany, February 1993. ONLINE GmbH. (cited on p. 49)

[Doberkat, 1992] E.-E. Doberkat. Integrating persistence into a set-oriented prototyping language. *Structured Programming*, 13(3):137–153, 1992. (cited on p. 49)

[Dourish, 1989] P. Dourish. A Transputer-based Parallel Lisp. Technical Report ECSP-TN-19, Edinburgh Concurrent Supercomputer Project, June 1989. (cited on p. 34, 142)

[Duke *et al.*, 1991] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z Specification Language: Version 1. Technical Report 91-1, University of Queensland, Software Verification Research Center, Queensland, Australia, January 1991. (cited on p. 5, 87, 95, 159)

[Duncan, 1990] R. Duncan. A survey of parallel computer architectures. *IEEE Computer*, pages 5–16, February 1990. (cited on p. 19, 141)

[Ekambareshwar and Downs, 1989] S. Ekambareshwar and T. Downs. Rapid prototyping of software systems using Prolog. In *Proc. Conference on Computing Systems and Information Technology*, pages 6–10, Sydney, Australia, August 1989. (cited on p. 150)

[Emerson, 1990] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier, 1990. (cited on p. 164)

[Faasen, 1991] C. Faasen. Implementing tuple space on transputer meshes. Master's thesis, University of the Witwatersrand, Johannesburg, South Africa, February 1991. (cited on p. 142, 144)

[Factor, 1990] M. Factor. The process trellis software architecture for real-time monitors. In *Proc. Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 147–155, Seattle, WA, March 1990. (cited on p. 80)

[Fleckenstein and Hemmendinger, 1989] C.J. Fleckenstein and D. Hemmendinger. Using a global name space for parallel execution of UNIX tools. *Communications of the ACM*, 32(9):1085–1090, 1989. (cited on p. 34, 142)

[Floyd, 1984] C. Floyd. A systematic look at prototyping. In Budde et al. [1984], pages 1–18. (cited on p. 6, 11, 13)

[Flynn, 1966] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966. (cited on p. 20, 141)

[Foster and Taylor, 1989] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, 1989. (cited on p. 25)

[Franke *et al.*, 1993] W. Franke, U. Gutenbeil, W. Hasselbring, C. Pahl, H.-G. Sobottka, and B. Sucrow. Prototyping mit Mengen — der ProSet-Ansatz. In H. Züllighoven, W. Altmann, and E.-E. Doberkat, editors, *Requirements Engineering '93: Prototyping*, volume 41 of *Berichte des German Chapter of the ACM*, pages 165–174. Teubner-Verlag, April 1993. (cited on p. 43)

[Fuchs, 1992] N.E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, September 1992. (cited on p. 150)

[Gabriel and McCarthy, 1988] R.P. Gabriel and J. McCarthy. Qlisp. In J.S. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*, pages 63–89. Kluwer Academic Publishers, 1988. (cited on p. 23)

[Gelernter and Philbin, 1990] D. Gelernter and J. Philbin. Spending your free time. *BYTE*, 15(5):213–219, 1990. (cited on p. 142)

[Gelernter *et al.*, 1985] D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel programming in Linda. In *Proc. International Conference on Parallel Processing*, pages 255–263, St. Charles, August 1985. IEEE. (cited on p. 33)

[Gelernter *et al.*, 1990] D. Gelernter, A. Nicolau, and D. Padua, editors. *Languages and Compilers for Parallel Computing.* Pitman, 1990. (cited on p. 184)

[Gelernter, 1984] D. Gelernter. A note on systems programming in Concurrent Prolog. In *Proc. International Symposium on Logic Programming*, Atlantic City, NJ, February 1984. (cited on p. 37)

[Gelernter, 1985] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985. (cited on p. 4, 31, 33, 75)

[Gelernter, 1989] D. Gelernter. Multiple tuple spaces in Linda. In *Proc. Parallel Architectures and Languages Europe (PARLE'89)*, volume 366 of *Lecture Notes in Computer Science*, pages 20–27. Springer-Verlag, June 1989. (cited on p. 81)

[Ghezzi and Jazayeri, 1982] C. Ghezzi and M. Jazayeri. *Programming Language Concepts.* Wiley, 1982. (cited on p. 51)

[Ghezzi *et al.*, 1991] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice-Hall, 1991. (cited on p. 150)

[Gimnich and Ebert, 1992] R. Gimnich and J. Ebert. Zur Definition und Interpretation ausführbarer Spezifikationen. In H. Boley, U. Furbach, and W.-M. Lippe, editors, *Sprachen für KI-Anwendungen — Konzepte, Methoden, Implementierungen*, pages 150–160. Münster (Schriftenreihe), 1992. (cited on p. 150)

[Goodenough, 1975] J.B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975. (cited on p. 47)

[Grama and Kumar, 1992] A.Y. Grama and V. Kumar. Parallel processing of discrete optimization problems: a survey. Technical report, University of Minnesota, Minneapolis, MN, November 1992. (cited on p. 16)

[Gray *et al.*, 1992] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, February 1992. (cited on p. 131)

[Gupta, 1990] A. Gupta, editor. *Proc. Europal/BCS-PPSG Workshop on High Performance and Parallel Computing in Lisp*, Twickenham, London, UK, November 1990. (cited on p. 185, 189)

[Halstead, 1985] R.H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985. (cited on p. 22, 52, 80)

[Hasselbring, 1990] W. Hasselbring. CELIP: A cellular language for image processing. *Parallel Computing*, 14(5):99–109, May 1990. (cited on p. 20, 36)

[Hasselbring, 1991a] W. Hasselbring. Combining SETL/E with Linda. In Wilson [1991b], pages 84–99. (cited on p. 114)

[Hasselbring, 1991b] W. Hasselbring. On Integrating Generative Communication into the Prototyping Language ProSet. Informatik-Bericht 05-91, University of Essen, December 1991. (cited on p. 114)

[Hasselbring, 1991c] W. Hasselbring. Translating a subset of SETL/E into SETL2. Informatik-Bericht 02-91, University of Essen, January 1991. (cited on p. 131, 140)

[Hasselbring, 1992a] W. Hasselbring. A Formal Z Specification of ProSet-Linda. Informatik-Bericht 04-92, University of Essen, September 1992. (cited on p. 5)

[Hasselbring, 1992b] W. Hasselbring. Programming cellular automata for image processing. Informatik-Bericht 01-92, University of Essen, February 1992. (presented at the BCS-PPSG Meeting on Cellular Automata, Imperial College, London, UK, February 12, 1992). (cited on p. 36)

[Hayes and Jones, 1989] I.J. Hayes and C.B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–338, November 1989. (cited on p. 150)

[Hazelhurst, 1990] S. Hazelhurst. A proposal for the formal specification of the semantics of Linda. Technical Report 1990-14, University of the Witwatersrand, Johannesburg, South Africa, October 1990. (cited on p. 4)

[Hazelhurst, 1991] S. Hazelhurst. A Linda solution to the evolving philosophers problem. *South African Computer Journal*, pages 44–53, September 1991. (cited on p. 71)

[Hekmatpour and Ince, 1988] S. Hekmatpour and D. Ince. *Software Prototyping, Formal Methods and VDM.* Addison-Wesley, 1988. (cited on p. 11)

[Hill *et al.*, 1992] M.D. Hill, J.R. Larus, S.K. Reinhardt, and D.A. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. In *Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 262–273, October 1992. (cited on p. 28)

[Hillis and Steele, 1986] W.D. Hillis and G.L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986. (cited on p. 20)

[Hiroyuki and Masaaki, 1991] S. Hiroyuki and S. Masaaki. Communication in Linda/Q — datatypes and unification. In *Proc. 20th International Conference on Parallel Processing*, volume II, pages 219–226, Boca Raton, FL, August 1991. CRC Press. (cited on p. 151)

[Hoare, 1985] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985. (cited on p. 2, 24)

[Horwat *et al.*, 1989] W. Horwat, A. Chien, and W. Dally. Experience with CST: Programming and Implementation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–109, June 1989. (cited on p. 22, 23)

[Hudak, 1986] P. Hudak. Para-functional programming. *IEEE Computer*, 19(8):60–70, 1986. (cited on p. 23)

[Hummel *et al.*, 1991] R. Hummel, R. Kelly, and S. Flynn Hummel. A set-based language for prototyping parallel algorithms. In *Proc. Computer Architecture for Machine Perception '91 Conference*, pages 135–146, Paris, France, December 1991. (cited on p. 20, 35)

[Hupfer, 1990] S. Hupfer. Melinda: Linda with multiple tuple spaces. Research Report 766, Yale University, New Haven, CT, February 1990. (cited on p. 81)

[Hutchinson, 1990] D. Hutchinson. Linda meets Lisp. In Gupta [1990]. (cited on p. 34, 76)

[Ito and Halstead, 1990] T. Ito and R.H. Halstead, editors. *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990. (cited on p. 23)

[Jagannathan, 1990] S. Jagannathan. Semantics and analysis of first-class tuple spaces. Research Report 783, Yale University, New Haven, CT, April 1990. (cited on p. 5)

[Jagannathan, 1991] S. Jagannathan. Customization of first-class tuple-spaces in a higher-order language. In *Proc. Parallel Architectures and Languages Europe (PARLE'91)*, volume 506 of *Lecture Notes in Computer Science*, pages 254–276. Springer-Verlag, June 1991. (cited on p. 34, 75, 81, 145, 153)

[Jellinghaus, 1990] R. Jellinghaus. Eiffel Linda: An object-oriented Linda dialect. *ACM SIGPLAN Notices*, 25(12):70–84, December 1990. (cited on p. 34)

[Jensen, 1990] K.K. Jensen. The semantics of tuple space and correctness of an implementation. Research Report 788, Yale University, New Haven, CT, April 1990. (cited on p. 4)

[Jensen, 1994] K.K. Jensen. *Towards a Multiple Tuple Space Model.* PhD thesis, University of Aalborg, Denmark, 1994. (in preparation). (cited on p. 4)

[Jones *et al.*, 1990] D.G. Jones, S.J. Dowdeswell, and T. Hintz. A rapid prototyping method for parallel programs. In T. Bossomaier, T. Hintz, and J. Hulskamp, editors, *The Transputer in Australasia (ATOUG-3)*, pages 121–128. IOS Press, 1990. (cited on p. 18)

[Jones, 1990] C.B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, 2nd edition, 1990. (cited on p. 4)

[Jozwiak, 1993] J.M. Jozwiak. Exploiting parallelism in SETL programs. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 1993. (cited on p. 20)

[Jul *et al.*, 1988] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988. (cited on p. 22)

[Kaashoek *et al.*, 1989] M.F. Kaashoek, H.E. Bal, and A.S. Tanenbaum. Experience with the distributed data structure paradigm in Linda. In *USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 175–191, Ft. Lauderdale, FL, October 1989. (cited on p. 74, 76)

[Kambhatla and Walpole, 1990] S. Kambhatla and J. Walpole. Recovery with limited reply: Fault-Tolerant processes in Linda. In *Proc. Second IEEE Symposium on Parallel and Distributed Processing*, pages 715–718, Dallas, December 1990. (cited on p. 83)

[Kane, 1991] A.J. Kane. A simple Linda-C parallel processing environment for symmetric multiprocessing VAX/VMS computer systems. Master's thesis, East Tennessee State University, December 1991. (cited on p. 141)

[Kemmerer, 1985] R. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, 11(1):32–43, 1985. (cited on p. 6)

[Kernighan and Ritchie, 1988] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, 2nd edition, 1988. (cited on p. 76)

[Korneev *et al.*, 1991] V.D. Korneev, N.N. Mirenkov, and A.S. Nepomniaschaya. Very-high-level language PARIS. In N.N. Mirenkov, editor, *Proc. International Conference on Parallel Computing Technologies*, pages 186–194, Novosibirsk, USSR, November 1991. (cited on p. 18)

[Krämer, 1991] B. Krämer. Prototyping and formal analysis of concurrent and distributed systems. In *Proc. Sixth International Workshop on Software Specification and Design*, pages 60–66. IEEE Computer Society Press, 1991. (cited on p. 18)

[Kruchten *et al.*, 1984] P. Kruchten, E. Schonberg, and J. Schwartz. Software prototyping using the SETL programming language. *IEEE Software*, pages 66–75, October 1984. (cited on p. 43)

[Kwiatkowska, 1989] M.Z. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7):371–386, 1989. (cited on p. 25, 61, 77)

[Landry and Arthur, 1992] K.D. Landry and J.D. Arthur. Instructional footprinting: A model for exploiting concurrency through instructional decomposition and code motion. Technical Report TR-92-35, Virginia Polytechnic Institute and State University, June 1992. (cited on p. 80)

[Lano and Haughton, 1992] K. Lano and H. Haughton. *The Z^{++} Manual.* Lloyds Register of Shipping, Croydon, UK, 1992. (cited on p. 5)

[Lawler and Wood, 1966] E.L. Lawler and D.E. Wood. Branch-and-bound methods: a survey. *Operations Research*, 14(4):699–719, July 1966. (cited on p. 65)

[LeBlanc and Mellor-Crummey, 1987] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, 1987. (cited on p. 152)

[Leichter, 1989] J.S. Leichter. *Shared tuple memories, buses and LAN's — Linda implementations across the spectrum of connectivity.* PhD thesis, Yale University, New Haven, CT, July 1989. (cited on p. 4, 75, 77, 78, 80, 83, 142, 143)

[Leler, 1989] W. Leler. PIX, the latest NEWS. In *Proceedings COMPCOM Spring '89*, San Francisco, February 1989. (cited on p. 34)

[Leler, 1990] W. Leler. Linda meets Unix. *IEEE Computer*, 23(2):43–54, 1990. (cited on p. 34, 80, 142)

[Levy and Pavlides, 1990] A.M. Levy and G. Pavlides. Simulation vs. prototype execution: a case study. In *Proc. International Conference on Computer Systems and Software Engineering (COMPEURO'90)*, pages 428–436, Tel Aviv, Israel, May 1990. IEEE Computer Society Press. (cited on p. 18)

[Li and Hudak, 1989] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989. (cited on p. 28)

[Litteck and Wallis, 1992] H.J. Litteck and P.J.L. Wallis. Refinement methods and refinement calculi. *Software Engineering Journal*, 7:219–229, May 1992. (cited on p. 119)

[Lucco, 1986] S.E. Lucco. A heuristic Linda kernel for hypercube multiprocessors. In *Proc. SIAM Conference on Hypercube Multiprocessors*, pages 32–38, September 1986. (cited on p. 142)

[Lucco, 1987] S.E. Lucco. Parallel programming in a virtual object space. *ACM SIGPLAN Notices (Proc. OOPSLA'87)*, 22(12):26–34, December 1987. (cited on p. 21)

[Lusk *et al.*, 1988] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D.H.D. Warren, A. Calderwood, P. Szeredi, S. Haradi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora Or-Parallel Prolog system. In Institute for New Generation Computer Technology (ICOT), editor, *Proc. International Conference on Fifth Generation Computer Systems*, volume 3, pages 819–830, Tokyo, Japan, 1988. Springer-Verlag. (cited on p. 23)

[Ma and Hintz, 1992] X. Ma and T. Hintz. A perspective on tools for distributed computation — background to the RE-Vision project. In *Proc. Fifth Conference of the NATUG group (NATUG-5).* IOS Press, 1992. (cited on p. 18)

[MacDonald, 1991] N. MacDonald. Linda work in Edinburgh. In Wilson [1991b], pages 100–104. (cited on p. 34)

[Markoff, 1992] J. Markoff. David Gelernter's Romance With Linda. *The New York Times (Business, section 3)*, page 1 and 6, January 19, 1992. (cited on p. 142)

[Matrone *et al.*, 1993] A. Matrone, P. Schiano, and V. Puoti. Linda and PVM: A comparison between two environments for parallel programming. *Parallel Computing*, 19(8):949–957, August 1993. (cited on p. 33)

[Matsuoka and Kawai, 1988] S. Matsuoka and S. Kawai. Using tuple space communication in distributed object-oriented languages. In *Proc. OOPSLA'88*, pages 274–284, San Diego, September 1988. (cited on p. 34, 80)

[McDowell and Helmbold, 1989] C.E. McDowell and D.P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, 1989. (cited on p. 152)

[Mehrotra and Rosendale, 1991] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. In Nicolau et al. [1991], pages 364–384. (cited on p. 19, 20)

[Mills *et al.*, 1991] P.H. Mills, L.S. Nyland, J.F. Prins, J.H. Reif, and R.A. Wagner. Prototyping parallel and distributed programs in Proteus. In *Proc. Third IEEE Symposium on Parallel and Distributed Processing*, pages 26–34, Dallas, TX, December 1991. (cited on p. 18)

[Milner, 1989] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989. (cited on p. 112)

[Murakami *et al.*, 1991] K. Murakami, O. Akashi, Y. Amagi, and H.G. Okuno. TOPS: A Tuple Operation Protocol Suite for NUE-Linda Computation Model. Technical report, NTT, Tokyo, Japan, 1991. (cited on p. 142)

[Mussat, 1992] L. Mussat. Parallel programming with bags. In Banâtre and Métayer [1992], pages 203–218. (cited on p. 27)

[Narem, 1989] J.E. Narem. An informal operational semantics of C-Linda V2.3.5. Technical Report 839, Yale University, New Haven, CT, December 1989. (cited on p. 4, 75, 78, 79)

[Naujokat, 1994] R. Naujokat. Entwurf und Implementierung einer Laufzeitbibliothek für ProSet-Linda auf einem lokalen Netzwerk. Master's thesis, University of Dortmund, 1994. (in preparation). (cited on p. 153)

[Nicolau *et al.*, 1991] A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors. *Advances in Languages and Compilers for Parallel Processing*. Pitman, 1991. (cited on p. 191)

[Nitzberg and Lo, 1991] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991. (cited on p. 28)

[O'Neill, 1992] G. O'Neill. Automatic translation of VDM specifications into Standard ML programs. *The Computer Journal*, 35(6):623–624, 1992. (cited on p. 150)

[Padua *et al.*, 1993] D.A. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin. Polaris: A new-generation parallelizing compiler for MPPs. CSRD Report No. 1306, University of Illinois at Urbana-Champaign, Urbana, IL, June 1993. (cited on p. 43)

[Parker, 1991] C.E. Parker. Z tools catalogue. Technical Report ZIP/BAe/90/020, British Aerospace, Warton, UK, May 1991. (cited on p. 5)

[Partsch, 1990] H.A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990. (cited on p. 14, 153)

[Patterson *et al.*, 1992] L.I. Patterson, R.S. Turner, R.M. Hyatt, and K.D. Reilly. Construction of a fault-tolerant distributed tuple-space. Technical report, University of Alabama at Birmingham, Birmingham, AL, 1992. (cited on p. 82, 83, 142, 144)

[Peskin and Segall, 1991] R.L. Peskin and E.J. Segall. Linda strategies for scientific computing environments. Technical Report CAIP-TR-137, Rutgers University, Piscataway, NJ, October 1991. (cited on p. 34, 144)

[Pinakis and McDonald, 1991] J. Pinakis and C. McDonald. The Inclusion of the Linda Tuple Space Operations in a Pascal-based Concurrent Language. In *Proc. 14th Australian Computer Science Conference*, Sydney, Australia, February 1991. (cited on p. 34)

[Pinakis, 1991] J. Pinakis. The Design and Implementation of a Distributed Linda Tuple Space. In *Proc. 2nd UWA Department of Computer Science Research Conference*, Nedlands, Australia, July 1991. (cited on p. 142, 144)

[Pinakis, 1992] J. Pinakis. Providing Directed Communication in Linda. In *Proc. 15th Australian Computer Science Conference*, Hobart, Australia, January 1992. (cited on p. 82)

[Pohland, 1994] H. Pohland. Entwurf und Implementierung eines graphischen Debuggers für ProSet-Linda. Master's thesis, University of Essen, 1994. (in preparation). (cited on p. 152)

[Pouget and Burkhart, 1993] J.-D. Pouget and H. Burkhart. Von Linda zu Linda$^+$. Technischer Bericht 93-1, University of Basel, IFI, Switzerland, March 1993. (cited on p. 34)

[Quinn and Hatcher, 1990] M.J. Quinn and P.J. Hatcher. Data-parallel programming on multicomputers. *IEEE Software*, 7(5):69–76, 1990. (cited on p. 20)

[Raina, 1992] S. Raina. Virtual shared memory: A survey of techniques and systems. Technical Report CSTR-92-36, University of Bristol, Bristol, UK, December 1992. (cited on p. 28)

[Reeves, 1991] A.P. Reeves. Parallel programming for computer vision. *IEEE Software*, 8(7):51–59, 1991. (cited on p. 20)

[Reisig, 1985] W. Reisig. *Petri Nets*. Springer-Verlag, 1985. (cited on p. 18)

[Roman and Cunningham, 1990] G.-C. Roman and H.C. Cunningham. Mixed programming metaphors in a shared dataspace model of concurrency. *IEEE Transactions on Software Engineering*, 16(12):1361–1373, 1990. (cited on p. 34)

[Schoinas, 1991] G. Schoinas. Linda and the Blackboard Model. In Wilson [1991b], pages 105–116. (cited on p. 75, 142)

[Schreiner, 1993] W. Schreiner. Parallel functional programming — an annotated bibliography. Technical Report 93-24, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1993. (cited on p. 23)

[Schwartz *et al.*, 1986] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets – An Introduction to SETL*. Springer-Verlag, 1986. (cited on p. 17, 43, 131)

[Sci, 1992] Scientific Computing Associates, New Haven, CT. *C-Linda User's Guide & Reference Manual*, 1992. (cited on p. 54, 76)

[Sewry, 1991] D.A. Sewry. A visual debugger/monitor for Linda tuple space. Technical Document PPG 91/14, Rhodes University, Grahamstown, South Africa, June 1991. (cited on p. 34)

[Shapiro, 1989] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989. (cited on p. 23–25, 37)

[Shekhar and Srikant, 1991] K.H. Shekhar and Y.N. Srikant. Linda Sub System on Transputers. In *Proc. Transputing'91*, pages 246–261, Sunnyvale, CA, April 1991. (cited on p. 142)

[Siegel and Cooper, 1991] E.H. Siegel and E.C. Cooper. Implementing Distributed Linda in Standard ML. Technical Report CMU-CS-91-151, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, October 1991. (cited on p. 34)

[Smith, 1991] N. Parker Smith. Network Linda achieves near-linear speed-ups on multi-system networks. *Supercomputing Review*, 4(5):53–54, May 1991. (cited on p. 34)

[Smith, 1992] G.L. Smith. A distributed implementation of the Linda virtual memory space on a cluster of processors. Technical Document PPG 92/11, Rhodes University, Grahamstown, South Africa, 1992. (cited on p. 142)

[Snyder, 1990] W.K. Snyder. The SETL2 programming language. Technical Report 490, Courant Institute, New York University, N.Y., September 1990. (cited on p. 131, 132, 136)

[Spivey, 1992a] J.M. Spivey. *The fUZZ Manual.* Computing Science Consultancy, Oxford, UK, 2nd edition, July 1992. (cited on p. 87, 95, 164, 165)

[Spivey, 1992b] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 2nd edition, 1992. (cited on p. 4, 5, 117, 119, 127, 159, 160, 164)

[Srini, 1986] V. Srini. An architectural comparision of dataflow systems. *IEEE Computer*, 19(3):68–88, 1986. (cited on p. 15)

[Stapleton, 1991] L. Stapleton. Grad student releases public-domain Linda. *Supercomputing Review*, 4(10):18, October 1991. (cited on p. 142)

[Stepney *et al.*, 1992] S. Stepney, R. Barden, and D. Cooper, editors. *Object Orientation in Z.* Springer-Verlag, 1992. (cited on p. 5)

[Stroustrup, 1986] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986. (cited on p. 164)

[Stumm and Zhou, 1990] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 23(5):54–64, May 1990. (cited on p. 28)

[Sun, 1990] Sun Microsystems, Inc. *Programming Utilities & Libraries*, 1990. (cited on p. 140)

[Sutcliffe and Pinakis, 1990] G. Sutcliffe and J. Pinakis. PROLOG-Linda: An embedding of Linda in muPROLOG. In *Proc. 4th Australian Conference on Artificial Intelligence (AI'90)*, pages 331–340, Perth, Australia, 1990. World Scientific, Singapore. (cited on p. 34, 151)

[Sutcliffe, 1993] G. Sutcliffe. Prolog-D-Linda v2: A New Embedding of Linda in SICStus Prolog. Technical Report 93/6, James Cook University of North Queensland, Townsville, Australia, January 1993. (cited on p. 34, 78, 151)

[Szymanski, 1991] B.K. Szymanski, editor. *Parallel functional languages and compilers.* Addison-Wesley, 1991. (cited on p. 23)

[Tanenbaum *et al.*, 1990] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990. (cited on p. 82)

[Tanenbaum, 1992] A.S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 1992. (cited on p. 81, 140)

[Thomas, 1991] O. Thomas. A Linda Kernel for UNIX Networks. In Wilson [1991b], pages 124–128. (cited on p. 142)

[Tolksdorf, 1992] R. Tolksdorf. Laura: A coordination language for open distributed systems. Report 1992/35, Technical University of Berlin, Berlin, Germany, 1992. (cited on p. 82)

[Trescher *et al.*, 1992] J. Trescher, F. Bieler, and C. Hinrichs. Modula-L: Implementation of the Linda Model for Arbitrary Transputer Networks. In W. Joosen and E. Milgrom, editors, *Parallel Computing: From Theory to Sound Practice (Proc. EWPC'92)*, pages 512–515, Barcelona, Spain, March 1992. IOS Press. (cited on p. 34, 142)

[Ushijima, 1989] D. Ushijima. Sharing supercomputer power. *MacWorld*, pages 83–85, June 1989. (cited on p. 142)

[Valentine, 1992] S.H. Valentine. Z^{--}, an executable subset of Z. In J.E. Nicholls, editor, *Z User Workshop, York 1991*, Workshops in Computing, pages 157–187. Springer-Verlag, 1992. (cited on p. 150)

[Wack, 1993] A.P. Wack. Scheme-Linda to C Compiler/Interpreter User Guide. CIS Tech-Report 93-23, University of Delaware, June 1993. (cited on p. 34)

[West and Eaglestone, 1992] M.M. West and B.M. Eaglestone. Software development: two approaches to animation of Z specifications using Prolog. *Software Engineering Journal*, 7(4):264–276, July 1992. (cited on p. 150)

[Wilson, 1991a] G. Wilson. Improving the performance of generative communication systems by using application-specific mapping functions. In *Proc. Workshop on Linda-Like Systems and Their Implementation* [1991b], pages 129–142. (cited on p. 153)

[Wilson, 1991b] G. Wilson, editor. *Proc. Workshop on Linda-Like Systems and Their Implementation.* Edinburgh Parallel Computing Centre TR91-13, June 1991. (cited on p. 80)

[Wilson, 1993] G. Wilson. *Structuring and Supporting Programs on Parallel Computers.* PhD thesis, University of Edinburgh, 1993. (cited on p. 74, 79–81)

[Wing, 1990] J.M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, pages 8–24, September 1990. (cited on p. 4, 150)

[Woodcock, 1991] J.C.P. Woodcock. A tutorial on the refinement calculus. In S. Prehn and W.J. Toetenel, editors, *VDM'91: Formal Software Development Methods*, volume 552 of *Lecture Notes in Computer Science*, pages 79–140. Springer-Verlag, 1991. (cited on p. 117, 140)

[Wordsworth, 1991] J.B. Wordsworth. The CICS application programming interface definition. In J.E. Nicholls, editor, *Z User Workshop*, Workshops in Computing, pages 285–294, Oxford, UK, 1991. Springer-Verlag. (cited on p. 5)

[Wyatt *et al.*, 1992] B.B. Wyatt, K. Kavi, and S. Hufnagel. Parallelism in object-oriented languages: A survey. *IEEE Software*, pages 56–66, November 1992. (cited on p. 21)

[Xu and Liskov, 1989] A.S. Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of Linda. In *Proc. 19th International Symposium on Fault-Tolerant Computing*, pages 199–206, June 1989. (cited on p. 83, 144)

[Yonezawa and Tokoro, 1987] A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming.* The MIT Press, 1987. (cited on p. 21)

[Yuen *et al.*, 1993] C.K. Yuen, M. D. Feng, W.F. Wong, and J.J. Yee. *Parallel Lisp Systems.* Parallel and Distributed Processing Series. Chapman & Hall, 1993. (cited on p. 34, 80)

[Zave and Schell, 1986] P. Zave and W. Schell. Salient features of an executable specification language and its environment. *IEEE Transactions on Software Engineering*, 12(2):312–325, February 1986. (cited on p. 18)

[Zave, 1984] P. Zave. The operational versus the conventional approach to software development. *Communications of the ACM*, 27(2):104–118, 1984. (cited on p. 6)

[Zenith, 1990] S.E. Zenith. Linda coordination language; subsystem kernel architecture (on transputers). Research Report 794, Yale University, New Haven, CT, May 1990. (cited on p. 142)

[Zenith, 1992] S.E. Zenith. A rationale for programming with Ease. In Banâtre and Métayer [1992], pages 147–156. (cited on p. 34)

[Zettler, 1992] P. Zettler. RISC Linda – A Dialect of Linda. Master's thesis, RISC-Linz, Johannes Kepler University, Linz, Austria, June 1992. (cited on p. 80)