On Integrating Generative Communication into the Prototyping Language ProSet

W. Hasselbring

Computer Science / Software Engineering
University of Essen
Germany
willi@informatik.uni-essen.de

December 1991

Abstract

ProSet is a procedural prototyping language based on the theory of finite sets. The coordination language Linda provides a distributed shared memory model, called tuple space, together with some atomic operations on this shared data space. Process communication and synchronization in Linda is also called generative communication, because tuples are added to, removed from, and read from tuple space concurrently. Synchronization is done implicitly.

In the presented proposal 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, specified fairness and selection for matching, and the facility for changing tuples in tuple space. 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 ProSet-Linda.

Contents

List of Figures

1 Introduction and overview

The set theoretic language ProSet[1] that is at present under development at the University of Essen is a successor to SETL [SDDS86, DF89]. The kernel of the language was at first presented in [DGH90b] and the system in [DGH90a]. Section 2 provides a brief introduction to the language.

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 building a model. Most systems in real life are of a parallel nature. Opportunities for automatic detection of parallelism in existing programs are limited and furthermore, in many cases the parallel formulation of a program is more natural and appropriate than a sequential one. The intention for integrating parallelism into a prototyping language is not only increased performance. However, parallel programming is conceptually harder 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. 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. The uncoupled and anonymous inter-process communication in Linda is in general not directly supported by the target machines. However, a *high-level* language must reflect a particular top-down approach to building software, not a particular machine architecture. This is also important to support portability across different machine architectures. Linda can be compared to explicit low-level parallel code such as message passing in much the same way as high-level programming languages can be compared to assembly code.

Section 3 will provide an introduction to the basic Linda model. In section 4 a proposal for combining ProSet with Linda will be presented. Section 5 sketches implementation issues and section 6 conclusions and future research directions for our work.

This report is an extended and revised version of [Has91a]. Major changes to this preliminary definition contain the process creator, derived from futures in Multilisp, which led to simplified depositing of tuples in tuple space and to a smoother combination between ProSet and Linda (sections 4.1 and 4.2.1). Matching may be customized (section 4.2.2), finding and changing of tuples in tuple space is merged into meeting tuples in tuple space (section 4.2.3), the fairness behavior for matching is specified in greater detail (section 4.2.4), and some examples were added (section 4.5).

Thanks are due to Robert Abarbanel, Henri Bal, Mike Factor, Michael Goedicke, Suresh Jagannathan, Jerry Leichter, and the participants of the Linda-Workshop in Edinburgh in June for some comments and discussions on various aspects of this work. The advice and comments on earlier drafts by E.-E. Doberkat are greatly appreciated.

2 The prototyping language ProSet

The following subsections will present a brief introduction to data and control structures of the language and two short examples. The high-level structures that ProSet provides qualify the language for prototyping. For a full account on prototyping with set-oriented languages we refer to [KSS84] or [DF89]. Persistence is supported by the language and discussed in [Dob90].

2.1 Data structures

ProSet provides the first-class data types `atom`, `integer`, `real`, `string`, `boolean`, `tuple`, `set`. It is a *higher-order* language, because functions and modules have first-class rights, too. First-class means to be expressible without giving a name. It implies having the right to be anonymous, being storable in variables and in data structures, having an identity, being comparable for equality with other objects, being returnable from or passable to a procedure. Each variable or constant is meant to be an *object*

[1] ProSet was formerly called SETL/E. The change was made to emphasize the prototyping aspect.

for our terminology. Both SETL and ProSet are weakly typed, i.e. the type of an object is in general not known at compile time. Integer, real, string and boolean objects are used as usual. Atoms are unique with respect to one machine and across machines. They can only be created and compared. The unary `type` operator returns a predefined type atom corresponding to the type of its operand.

Atom, integer, real, string, boolean and function objects are basic, because they are not built from other objects. Tuples and sets are compound data structures, which may be heterogeneous composed of basic data objects, tuples, and sets. Sets are unordered collections while tuples are ordered. The following expression 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 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 e.g. selection of an element from an empty set. `Om` itself may not be element of a set, but of a tuple.

2.2 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 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 at first. The quantifiers ($\exists$, $\forall$) 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 provided an existentially quantified expression with the same iterator would yield `true`. 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.

```
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 F1(2) /= F2(2) and
                      (abs(F2(1)-F1(1)) /= abs(F2(2)-F1(2)))));
   end NonConflict;

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

   procedure abs (i);
   begin
      return if i >= 0 then i
                       else -i
             end if;
   end abs;
end Queens;
```

Figure 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`. `T(i)` selects the `i`th element from tuple `T`.

2.3 Examples

In Fig. 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. 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 positions 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. 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 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 | (notexists 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);
      nonleafs := domain (edges);       -- {1,2,3,"a","b"}
      -- Successively remove nonleafs that don't point to nonleafs:
      whilefound x in nonleafs | (edges{x} * nonleafs = { })
      do
         nonleafs less:= x;
      end whilefound;
      return (nonleafs /= { }) ;
   end ContainsCycle;
end TopSort;
```

Figure 2: Topologically sorting the nodes of a directed graph.

As sets are unordered collections, the program may print the fields and positions in different sequences. 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 specification of the queens' problem.

The program in Fig. 2 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 binary operators `with` and `less` add elements to resp. remove elements from sets and tuples. `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 "`*`" resp.. One possible result would be the following tuple in `SortTup`:

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

3 Linda

Linda[2] is a coordination language concept for explicitly parallel programming in an architecture independent way, which has been developed by David Gelernter at Yale University [Gel85]. Communication in Linda is based on the concept of tuple space, i.e. a virtual common data space accessed by an associative addressing scheme. [CG90] provides a full account to parallel programming in Linda. A comparison with other approaches to parallel programming may be found in [CG89].

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 interrelations explicitly. The parallel processes are decoupled in time and space in a very simple way. 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 and a natural synchronization paradigm and at the same time 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 (section 2). 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 also 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 placeholder for such a value. A formal is prefixed with a question mark. Tuples in tuple space are selected by a matching procedure, 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.

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:[3]

out (tuple); The specified tuple is evaluated and then added to the tuple space. The `out`-executing process continues as soon as evaluation of the tuple is completed. The "`out("data", 123);`" operation in Fig. 3 deposits the tuple `["data",123]` into tuple space.

No specified action is taken in the event that tuple space is full. See also sections 4.2.1, 4.4.1, and 4.5.1 for discussions on this subject.

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

[2] Linda is a registered trademark of Scientific Computing Associates.
[3] Parts of this description were derived from [CG90, appendix A].

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 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.

The main program is the only process that lives outside of tuple space.

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 whole. 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 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 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.

inp (template) / rdp (template) In C-Linda, 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 section 4.4.2).

A tuple and a template match, iff

- the tuple is passive, and
- the numbers of fields are equal, and
- 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.

Some approaches allow formals in deposited tuples. Such formals match with appropriate actuals in templates, but never with formals in templates (see also the discussion on formals in section 4.4.4).

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 section 4.5.1 for an example. 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* [BCG+91].

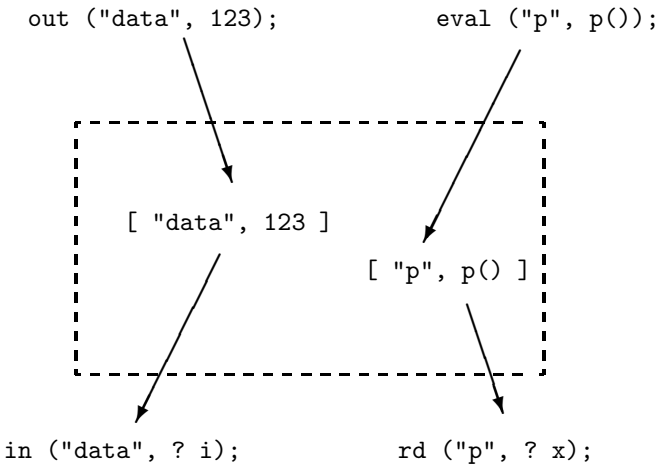


Figure 3: Tuple-space communication in Linda.

A *parallel programming language* consists of a coordination language like Linda and a sequential computation language like C [CG92]. The first computation language, in which Linda has been integrated, was C [Gel85, ACG86]. Implementations of C-Linda have been performed on a wide variety of parallel architectures: shared-memory multi-processors like the Sequent Balance and Symmetry, and the Encore Multimax [Car87] as well as on distributed memory architectures such as the S/Net [Car87], Hypercubes [Luc86, BCG89], Parwell [BHK89], Transputers [Dou89, Zen90, Faa91, CCH91], and Ethernet [WL88, AB89, SP90, CKM91, Sch91]. Especially in the Yale Linda Implementation extensive compiler optimizations are applied [CG91]. Also hardware support has been considered [ACGK88]. Recent applications experience is discussed in [BCG+91].

Meanwhile there exist also integrations into higher-level languages such as Modula [BHK89], C++ [CCH91], Smalltalk [MK88], Eiffel [Jel90], Lisp [Hut90, YW90, Aba91], EuLisp [BP91], Scheme [Dou89, DM90, Jag91], Russell [But91] or Prolog [SP90, Cia91, Mac91], and for operating systems [Lel90]. Combinations with other languages are planned [Mac91].

4 Generative communication in ProSet

A *coordination language* like Linda provides means for process creation and inter-process communication which may be combined with *computation languages* like SETL [CG92]. The following subsections will discuss process creation and tuple-space communication in ProSet, some design alternatives, and some small examples. We regard tuple spaces primary as a device for synchronization and communication between processes, and only secondary for process creation.

4.1 Process creation

In Linda there is 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 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.

4.1.1 Multilisp's futures

Multilisp [Hal85] 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 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, e.g. arithmetic, comparison, type checking, etc., touch their operands. This is opposed to simple *transmission* of a value from one place to another, e.g. by assignment, passing as a parameter to a procedure, returning as a result from a procedure, building the value into a data structure, which does **not** touch the value. Transmission can be done without waiting for the value.

An analogous approach may be found in Qlisp [GM88] where the propositional parameter EAGER lets the functions QLET and QLAMBDA evaluate their arguments in parallel like Multilisp's futures do. The semantics of futures is based on *lazy evaluation*, which means that an expression is not evaluated until its result is needed. I-Structures in Id [ANP89] provide special kinds of arrays, whose components are evaluated in parallel. Access to an array component is suspended until the requested component is evaluated: the array is not touched as a whole. BaLinda Lisp [YW90] provides also a notion for *futures*, but this is like a fork in Unix[4]. A *touch* does not guarantee a deterministic result, because blocking is not applied when a future is touched. The *future* concept is usually implemented on shared-memory multi-processors.

4.1.2 Process creation in ProSet

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. In our approach the concept for process creation via futures is adapted to set-oriented programming and combined with the concept for synchronization and communication using tuple spaces.

Multilisp is based on Scheme, 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*.

[4] Unix is a registered trademark of AT&T.

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 immediately 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
```

After statement 1 is executed the 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.

Also, 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 you 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 [Hal85, page 511].

Conversely, in iterative formers such as "`{ || p() .. || q() }`" the *dot* operator "`..`" needs the initial and final values for the iteration, and thus has to wait for the termination of `p()` and `q()`.

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 I-Structures in Id [ANP89] where access to array components touches only the requested components. It is not really necessary for tuples to touch them as a whole, but to handle compound data structures as in Id might cause problems for selection from sets. We decided to touch compound data structures always as a whole to retain consistency for tuples and sets.

The actual parameters for the procedure `doit` in

```
x := doit ( || p(), || q() );
```

are evaluated concurrently to each other. The procedure `doit` is invoked when both processes have terminated their execution. The programmer of `doit` does not have to know that the actual parameters are evaluated concurrently. The runtime system takes care of that. Returning an expression that is prefixed by `||` achieves concurrency according to the context of the corresponding procedure invocation.

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, passing as a parameter to a procedure,

returning as a result from a procedure, and depositing in tuple space (this is discussed in section 4.2.1). 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. The unique moment at which a value is not touched is on creation of the respective process, because this is the unique moment at which no copy is needed.

Analogously to statements, concurrency is achieved in declarations like

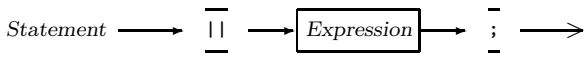
```
constant c := || p();
visible x := [ || p() ];
```

If, after one of such a declaration or similar statement, `x` is assigned a new value, then the corresponding spawned process will be abandoned[5] provided it is still running. Hence, the automatic garbage collection provides a means for explicit process termination outside tuple space. The automatic garbage collection also should take place when the existence of an object terminates. This is the case e.g. on return from a procedure or on program termination. Automatic garbage collection does not apply to processes within tuple space. Process termination within tuple space is discussed in section 4.3.

Process-spawning statements Also the following statement, which spawns a new process, is allowed:

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

The return value of such a process will not be available and it is not possible to abandon it explicitly. The general form of such a process-spawning statement is



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 that compute their operands. For instance, in the following expressions the return values are needed:

```
   1 + || p()
["x"] + [|| p()]
    - || p()
  type || 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. The following expression spawns three processes:

```
|| { || p(), 123, || q() }
```

The set-forming process has to wait for the termination of `p()` and `q()`.

Side effects and write parameters are not allowed for processes. Communication and synchronization is done only via tuple-space operations. However, processes may access common, persistent data objects via critical regions on these objects. This is discussed elsewhere [Dob90].

[5] Killing processes has to be done with care. Especially when such processes are still doing tuple-space operations.

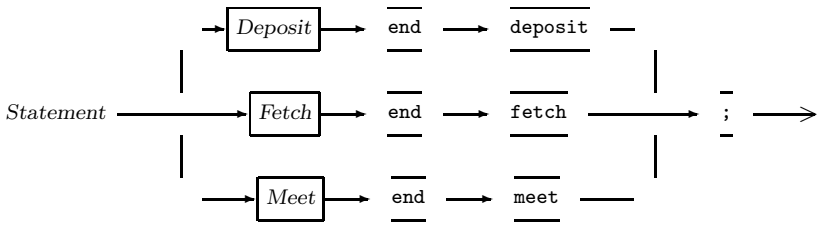
Notice that processes have **no** first-class rights in ProSet (see also section 4.4.4). The `type` operator has to wait for termination as any other operator.

The process creator may also be inserted automatically or semi-automatically by a parallelizing compiler, which detects implicit parallelism (probably fine-grained). This is especially interesting in the context of *transformational programming* [Par90].

Historical note In [Has91a] `||` was only a syntactical marker within tuple-space operations. By means of promoting `||` to an unary operator we gain additional flexibility and smoother semantics for the tuple-space operations.

4.2 Tuple-space operations

ProSet provides three tuple-space operations:



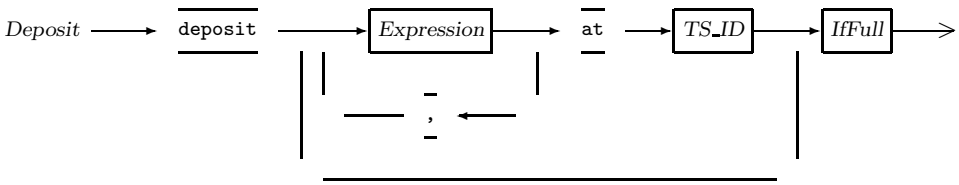
The `deposit` operation deposits new tuples into tuple space, the `fetch` operation fetches and removes a tuple from tuple space, and the `meet` operation meets and leaves a tuple in tuple space. It is possible to change the tuple's value while meeting it.

There is no difference between ProSet-tuples and Linda-tuples. 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 e.g. via maps from tuples to counts, but this would not reflect the matching provided by tuple spaces. We will return to this question when discussing the introduction of a type "*tuple space*" in section 4.4.4.

Tuple-space operations are statements that yield **no** values. They should not be confused with operators in expressions that always yield values.

4.2.1 Depositing tuples

The `deposit` operation deposits tuples into a specified tuple space:



It is possible to deposit several tuples in an expression list into one tuple space and several such expression lists into multiple tuple spaces by one statement, but there are no guarantees made for the chronological order of availability of these tuples for other operations that wait for them (see

below). The tuples are handed over to the tuple-space manager, which adds them to the tuple space in an arbitrary order. There is no guarantee given for the time of availability of deposited tuples for matching templates (see also sections 4.2.4 and 4.4.2).

All expressions are evaluated in arbitrary order, before any tuple is put into tuple space. The expressions must yield tuples to be deposited in tuple space; if not, an exception will be raised. The identifier *TS_ID* will be discussed in section 4.3.

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 (cp. `out` of C-Linda). If there are executing processes in a tuple, then this tuple is added as an active one to tuple space. Depositing a tuple into tuple space does not touch the value. 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.

Depositing in tuple space does not touch the values of tuples to be deposited thus the following statement deposits an *active* tuple into tuple space:

```
deposit [ || p() ] at TS
```

whereas the following statement sequence deposits an *passive* tuple:

```
x := [ || p() ];
deposit x at TS;
```

The `deposit` operation copies the value of `x`, and thus has to wait for the termination of `p()` (see also section 4.1.2).

Limited tuple spaces

Because every existing computing system has only finite memory, the memory for tuple spaces will also be limited. Pure tuple-space communication does not deal with *full* tuple spaces: there is always enough room available. Thus most runtime systems for Linda hide the fact of limited memory from the programmer. For instance, in [Hut90] `out` is suspended until space is available again. Conversely, programs written in C-Linda from Scientific Computing Associates print the message "`out of tb's`" and exit when the tuple space is exhausted [She90].

In ProSet-Linda the predefined exception `TS_IS_FULL` will be raised by default when no memory is available for a `deposit` operation. If there are multiple tuples specified in one `deposit` operation, then none of them has been deposited when `TS_IS_FULL` is raised. Conversely, this exception will be raised, if at least one tuple cannot be deposited in any tuple space.

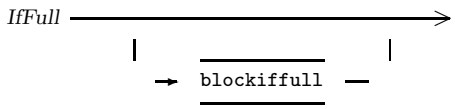
In ProSet it is possible to specify a handler for an exception by annotating a statement with a new binding between exception name and handler name:

```
deposit [ x ], [ || f(x) ]
     at TS
end deposit when TS_IS_FULL use MyHandler;
```

If not explicitly specified with the ProSet-`stop` statement, the user-defined handler will not abort the program. If the handler executes a `return` statement, then the statement following the `deposit` will be executed and none of the tuples of the respective `deposit` will be deposited. If the handler executes a `resume` statement, then the `deposit` operation tries again to deposit the tuples. The programmer has to take care not to produce an infinite alternation between raising and resuming. See

also section 4.5.1 for an example. If no user-defined handler is given, then the runtime system will abort the program with an error code.

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

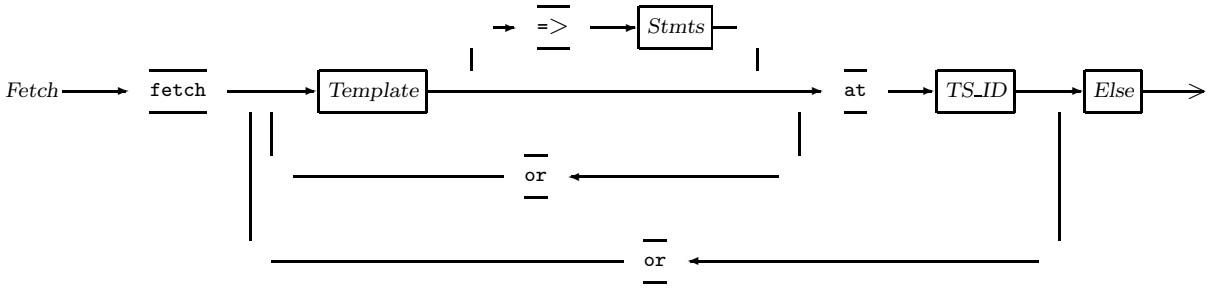


You may specify `blockiffull` as well as a new binding between the predefined exception `TS_IS_FULL` and another handler name, but then `TS_IS_FULL` will never be raised. The compiler will print a warning message in this case.

The suitable handling of full tuple spaces depends on the application you have to program. Thus a general setting does not seem to be appropriate. Blocking is useful e.g. in a producer-consumer application. In a master-worker application you might prefer to collect some results by your own handler before producing more tasks, when your tuple space is full. If you would block both, master and worker, a deadlock might occur, in contrast to the meaning of pure tuple-space communication. See also section 4.4.1 for a discussion on this subject. In section 4.5.1 a master-worker application with limited tuple spaces will be presented.

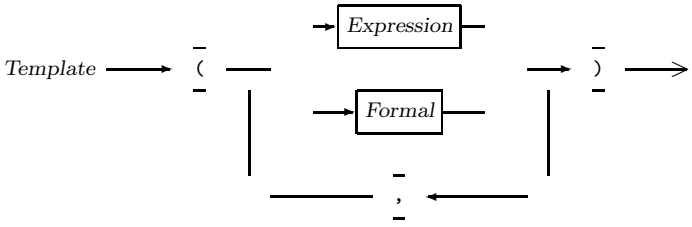
4.2.2 Fetching tuples

A `fetch` operation fetches and removes one tuple from a tuple space:

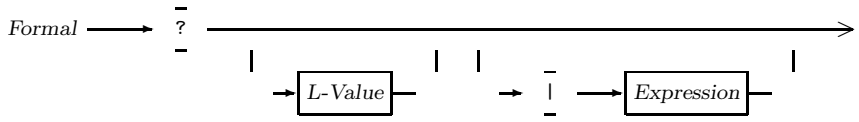


It is possible to specify several templates for multiple tuple spaces in one statement, but only one template may be selected nondeterministically (section 4.2.4). If there are no `else` statements specified (see below) then the statement suspends until a match occurs. The selected tuple is removed from tuple space. If statements are specified for the selected template, these statements are executed (only for this template).

A template consists of a list of ordinary expressions and so-called *formals*:



The expressions are called *actuals*. They are at first evaluated in arbitrary order. The list is enclosed in parentheses and not in brackets in order to set the templates apart from tuples. The fields that are preceded by a question mark are the *formals* of the template:



As usual `|` means *such that*. The Boolean expression behind `|` may be used to customize matching. Several approaches to customizing the matching process were proposed. E.g. [Sch91] and [AS91] provide *range accessing*: formals are extended to match only values in a specified range. [Aba91] provides so-called *satisfies forms*, viz. *predicate functions*, which are attached to formals to be evaluated during the matching process. [BP91] allows to redefine matching methods. In [Jag91] individual tuple spaces are accessed indirectly via so-called *policy closures*, which may customize the matching process.

A tuple and a template match, iff all the following conditions hold:

- the tuple is passive (matching touches the value)
- the numbers of fields are equal
- types and values of actuals in templates are equal to the corresponding tuple fields
- the Boolean expressions behind `|` in the formals evaluate to `true`. If no such expression is specified in a formal, then this field matches unconditionally

The *L-Values* specified in the formals are assigned the values of the corresponding tuple fields provided matching succeeds. If such *L-Values* occur in the expressions of formals, then these expressions are evaluated with the corresponding *old* values of these *L-Values*. The question mark may be used like an expression as a placeholder for the value of the corresponding tuple field:

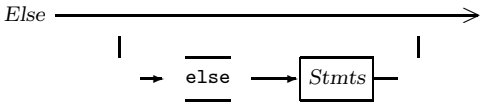
```
fetch ( "x", ? x |(type ? = integer) ) at TS end fetch;
```

This formal only matches integer values in the corresponding tuple field. Conversely, the formal "`? x |(type x = integer)`" would test the value of `x` *before* the assignment of the corresponding tuple value takes place. It is only allowed to use the question mark this way in expressions that are parts of formals. It is not possible to access the values of multiple tuple fields within expressions of formals.

If a *L-Value* is specified more than once, it is not determined which of the possible values is assigned. If there is no *L-Value* specified, the corresponding value will not be available. You may regard a formal without an *L-Value* as a "don't care" or "only take care of the condition" field.

Non-blocking matching

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



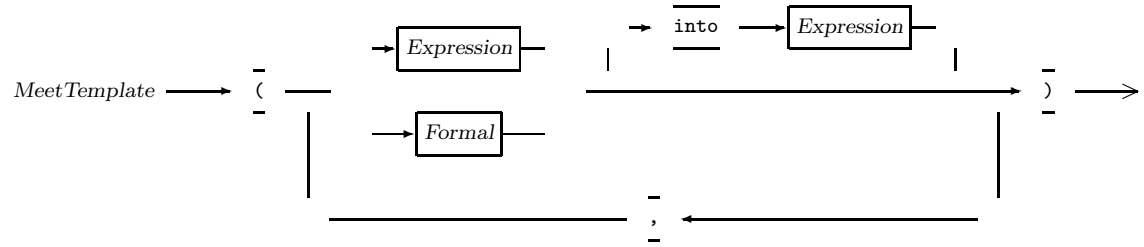
We will use the notion *non-blocking matching* if `else` statements are specified as opposed to *blocking matching* if no `else` statements are specified. In sections 4.4.2 and 4.5.1 discussions on this subject may be found.

An example for the `fetch` operation:

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

4.2.3 Meeting tuples

The `meet` operation meets and leaves one tuple in tuple space. It is possible to change the tuple while meeting it. Exchanging the keyword `fetch` with `meet` and the nonterminal *Template* with *MeetTemplate* in the first syntax diagram of section 4.2.2, one obtains the syntax for the `meet` operation:



The expressions are evaluated as usual, the formals are used to create templates, which are used for matching as with the `fetch` operation (section 4.2.4). 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).

An example for the `meet` operation:

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

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. Except for the fact that the `meet` operation without `into`'s leaves the tuple it found in tuple space, it works like the `fetch` operation.

Changing tuples

The absence of support for user-defined high-level operations on shared data in Linda is criticized [Bal91]. 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 ( "x", ? |(type ? = integer) into ?+1 ) at TS end meet;
```

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

```
fetch ( "x", ? x |(type ? = integer) ) at TS end fetch;
deposit [ "x", 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 another process, which does simultaneously a non-blocking `meet` operation with the same template, may execute its `else` statements.

With the `meet` operation expensive copying of compound data may be avoided (see also section 5). The question mark may be used like an expression as before. It is not necessary to specify *L-Values* for changing tuple fields. However, they may be used:

```
meet ( "x", ? x |(type ? = integer) into ?+1 ) at TS end meet;
```

Where `x` is assigned the value of the corresponding tuple field *before* the change takes place. Remember that it is only allowed to use the question mark this way in expressions that are parts of formals.

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 section 4.3). The place for the met tuple will be reserved for the whole operation. However, if the changed tuple exceeds a physical memory limit, this will raise `TS_IS_FULL`.

Historical note In [Has91a] we presented a `find` and a `change` operation. While refining the `change` operation we recognized that it contained the `find` operation and consequently we merged both into the `meet` operation.

4.2.4 Nondeterminism and fairness while matching

There are two sources for nondeterminism while matching:

1. Several matching tuples exist for the templates: one tuple will be selected nondeterministically.
2. The selected 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 [Kwi89]:

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.

Since deposited tuples are no longer connected with processes, it is reasonable to select them without any consideration of fairness. Linda's semantics do 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. The system is enabled to store the tuple space e.g. in a hash-based way. It is not necessary to retain the *age* of a tuple in any way.

To specify weakly fair selection of templates by means of temporal logic we introduce the following abbreviations for predicates on a process P and a template T:

E	=	"Process P executes a blocking `fetch` or `meet` operation with template T"
M	=	"There is a matching tuple for template T in tuple space"
B	=	"Process P is blocked with template T"
S	=	"Template T is selected for a matching tuple provided there exists one"
A	=	"Process P is activated (template T was selected)"

Now the above-given fairness guarantee may be formulated as follows:[6]

$$\mathbf{E} \wedge \mathbf{M} \wedge \mathbf{S} \Rightarrow \mathbf{A} \quad (1)$$

$$\mathbf{E} \wedge \neg(\mathbf{M} \wedge \mathbf{S}) \Rightarrow \mathbf{B} \quad (2)$$

$$\mathbf{B} \wedge \Box\Diamond\mathbf{M} \Rightarrow \Diamond(\mathbf{S} \wedge \mathbf{A}) \quad (3)$$

Predicates 1 and 2 describe the behavior on executing blocking `fetch` or `meet` operations. Predicate 3 describes the selection of a template that belongs to a suspended process. "$\Box\Diamond\mathbf{M}$" means that there is infinitely often a matching tuple available for the template. **S** (selection) is implied by "$\Box\Diamond\mathbf{M}$".

Weakly fair selection of templates applies only to blocking matching: if a template that 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 (resp. processes), which are suspended because no tuple matches them are weakly fair matched with tuples later deposited. The implementation has to guarantee this (see also section 5).

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. This does not seem to be acceptable, since non-blocking matching already implies some vaguenesses (section 4.4.2). You have to be aware that busy waiting with polling methods, which use non-blocking matching operations e.g. in loops, are **not** handled in a fair way.

[6] In temporal logic "$\Diamond p$" and "$\Box p$" mean that predicate p holds eventually resp. always. See e.g. [Eme90] for a full account to temporal logic.

4.2.5 Comparison with the C-Linda operations

The `deposit` operation comprises the `out` and `eval` operations of C-Linda. You might compare depositing of active tuples with `eval`, but it is not exactly the same, however, because all fields of an `eval` tuple are executed concurrently 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 concurrently. But probably no system will create a new process to compute e.g. a plain integer constant. In the Yale Linda Implementation, only expressions consisting of a single function call are evaluated within new processes [CG90]. The system has to decide, which fields to compute concurrently and which sequentially. Similar problems arise in automatic parallelization of functional languages: here you have to reduce the existing parallelism to a reasonable granularity. In our approach the programmer has to communicate his knowledge about the granularity of his application to the system.

Additionally, the semantics of `eval` is not always well 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.

The `fetch` operation merges `select` of Ada and `in` resp. `inp` of C-Linda. The `meet` operation merges `select` of Ada, `rd` and `rdp` of C-Linda, and allows for changing tuples in tuple space.

4.3 Multiple tuple spaces

Atoms are used to identify tuple spaces. As mentioned in section 2 atoms are unique for one machine and across machines. They have first-class rights. We considered to introduce a new data type for tuple-space communication, but do not see a justification for the time being (see also section 4.4.4).

ProSet provides several library functions to handle multiple tuple spaces:

`createTS(limit):` Calls the 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`. Thus you can only use atoms that were created by `createTS` to identify tuple spaces.

Since one has exclusive access to a fresh assigned tuple-space identity, `createTS` provides 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.

In C-Linda from Scientific Computing Associates the size for the global tuple space is specified in byte blocks [She90]. However, the level of such an unit is too low for a prototyping language.

`existsTS(TS):` Provides `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 should be indivisible for `TS`.

This function appears to be useful e.g. in a master-worker application: when the work has been done, the master can remove garbage and abandon the workers.

`removeTS(TS):` Calls `clearTS(TS)` and removes `TS` from the list of existing tuple spaces. Note that after assigning a new value to a variable that contains a tuple-space identity, the respective tuple space is **not** removed. The tuple-space identity is garbage collected, not the tuple space itself. Garbage collection applies only to first-class objects.

If functions are invoked with actual parameters of wrong type, then an appropriate exception will be raised.

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. If all processes are suspended, the tuple-space manager should terminate with an appropriate warning message. A concurrent program terminates when all its sequential processes have terminated.

Tuple space communication in ProSet as presented in this paper is designed for *multiprocessing* (single application running on multiple processors) as opposed to *multiprogramming* (separate applications). Multiprogramming is done via a separate mechanism for handling persistent data objects, which is based on critical regions.

4.4 Discussion of some design alternatives

4.4.1 Limited tuple spaces

The meaning of raising `TS_IS_FULL` 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. We will return to this subject in the example of section 4.5.1.

4.4.2 Non-blocking matching

Especially in a distributed environment there may be matching tuples available while the `else` statements of a non-blocking `fetch` or `meet` operation are executed. 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. Consider the following statement sequence:

```
deposit [ "foo" ] at TS end deposit;
fetch ( "foo" ) at TS
  else put ("Not yet deposited or already fetched by another one");
end fetch;
```

Remember that there is no guarantee given for the time of availability of deposited tuples for matching templates (section 4.2.1). It is recommended not to use non-blocking matching, if such vagueness is a problem.

Historical note In [Has91a] we did not support non-blocking matching because of the mentioned problems. However, despite this vagueness we decided to provide non-blocking matching for practical reasons: we understand that the semantic problems with supporting the notion of limited tuple spaces and non-blocking matching are related. In section 4.5.1 we will return to these subjects.

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

But it is intuitively not obvious to us, whether both templates have to match with tuples at the same time or during an interval of time (between starting and finishing the `meet` or `fetch` operation). E.g. in [AS91] multiple templates are allowed for `rd` and `rdp` operations, which set read-locks on the being met tuples to guarantee simultaneous matching. Conversely, in [BP91] accumulate matching for multiple templates is discussed, where matching must take place between starting and finishing the respective operation. Due to the unclear semantics we decided not to support template lists; the above statement is not allowed in ProSet.

4.4.4 Introducing new data types

In this section we shall discuss the introduction of new data types for generative communication in ProSet. As Linda relies heavily on type matching, the type system of the computation language has a notable effect on tuple-space implementation and semantics. E.g. in C the equivalence of types is not that obvious. Under which conditions are structures resp. unions equivalent? Are pointers equivalent to array-names? In [Lei89] 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 Linda-tuples, a combination on the basis of this common feature becomes straightforward. However, it is necessary to discuss the introduction of *additional*, first-class data types to support generative communication in ProSet.

Processes and process identities

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 equal 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. 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." ([Hal85, page 505])

We decided not to sacrifice this simplicity.

Tuple spaces and tuple-space identities

The idea of splitting the tuple space into multiple spaces is frequently applied. New data types resp. classes are often proposed to organize them [MK88, Lel90, Aba91, BP91, But91, Cia91, Sch91, and others]. In the basic tuple-space operations values of objects of these types are 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 [Jag91] the function `make-ts`

returns a reference to a *representation structure* of a newly created tuple space. Group identifiers are used in [And91a] 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 (section 3).

In [Gel89] and [Hup90] 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 (e.g. 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 ..." ([Hup90, page 12])

Conversely, [Cia91] 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. For the time being, we see no necessity to introduce a *new* data type with first-class rights for tuple-space identities.

Formals and templates

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.

4.5 Examples

We will now present the solutions for a master-worker application with limited tuple spaces, the queens' problem, parallel matrix multiplication, and the dining philosophers problem.

4.5.1 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 meaning of non-blocking matching in correlation with limited tuple spaces.

The program in Fig. 4 consists of a master (the main program), a worker procedure, and an exception handler. The master spawns `20` worker processes outside the tuple space. These workers execute in an infinite loop, in which tasks are fetched from tuple space `WORK`, and results are computed and deposited in tuple space `RESULT`. A worker is suspended if there is no more space available for depositing results, since it does not know what to do in this case. The master repeatedly deposits tasks into `WORK`. If there is no more space available for more tasks, the exception `TS_IS_FULL` will be raised and the handler `fetchResults` will be activated to fetch some results. At least one result will be fetched and probably some more, before resuming the `deposit` operation. It is not important *how many* results are fetched. It is only important that *some* results are fetched to enable the workers. Hence, the vaguenesses introduced by non-blocking matching make no problems in such applications.

```
program MasterWorker;
  visible constant WORK := createTS (1000),
                   RESULT := createTS (500);
  visible tasks := { set of some task tuples };
begin
    for i in [1.. 20] do
        || 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 quit; -- quits the infinite loop
            end fetch;
            -- do something with the result
        end loop;
        tasks +:=  {a set of some new task tuples dependent on the results};
        resume;  -- the deposit operation
    end fetchResults;
end MasterWorker;
```

Figure 4: A master-worker program with limited tuple spaces.

Notice that matching does not fail while executing "`else quit;`": matching fails *before* the `else` statements are executed.

The program would be 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 concurrently running workers. However, the workers access global constants.

As you can see in the example, we found non-blocking matching useful to handle limited tuple spaces, because we did not know the exact number of results we had to fetch. We have reasons to believe that the semantic problems with supporting the notion of limited tuple spaces (section 4.4.1) and non-blocking matching (section 4.4.2) are related:

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

The specification of a limit for a tuple space is primary intended here to obtain a facility for load balancing between a producer and a consumer. Remember that raising `TS_IS_FULL` does not distinguish between reaching the specified size or reaching a physical memory limit.

If you regard limited tuple spaces as a device to influence load balancing (not necessary load balancing between hardware processors, but between processes), then the vagueness introduced by limited tuple spaces and non-blocking matching is no problem: you deposit *some* tasks, thereupon you fetch *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 vaguenesses might be a problem, you should not use non-blocking matching, not specify limits for tuple spaces, and not customize the behavior on reaching full tuple spaces.

4.5.2 The queens' problem revisited

In section 2.3 the queens' problem was introduced together with a sequential solution. In Fig. 5 a parallel solution based on the master-worker model is given. It is recommended to examine the sequential solution in Fig. 1 on page 3 again.

The resulting set of non-conflicting positions is built 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 master program uses explicit loops, whereas in the sequential program no explicit loops were needed and thus the parallel solution seems to have a lower level of abstraction than the sequential one. Such observations are often made in a *wide spectrum language*[7] like ProSet, where programs may be transformed within the language using lower-level constructs to increase efficiency. For a full account to program transformation techniques we refer to [Par90].

[7] In a *wide spectrum language* it is possible to program on a high level of abstraction as in Fig. 1 as well as on a level of e.g. Pascal [Par90].

```
program ParallelQueens;
   constant N := 8, NumWorker := 20;
   visible constant WORK := createTS(om),
                    RESULT := createTS(om);
begin
   Positions := npow(N, {[x,y]: x in [1..N], y in [1..N]});

   for i in  [1 .. NumWorker] do
      deposit [ || Worker() ] at WORK end deposit;
   end for;
   deposit [ {} ], [ 0 ] at RESULT end deposit;  -- initialize the result set
   for NextPosition in Positions do
      deposit [ NextPosition ] at WORK end deposit;
   end for;

   fetch ( #Positions ) at RESULT end fetch;
   fetch ( ? NonConflict | (type ? = set) ) at RESULT end fetch;
   put (NonConflict);
   ClearTS(WORK);

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

         if forall F1 in MyPosition, F2 in MyPosition |
                        (F1 /= F2 !implies
                          (F1(1) /= F2(1) and F1(2) /= F2(2) and
                          (abs(F2(1)-F1(1)) /= abs(F2(2)-F1(2)))))
             then
                meet ( ? |(type ? = set) into (? with MyPosition) )
                    at Result
                end meet;
         end if;
         meet ( ? |(type ? = integer) into (? + 1) )
            at Result
         end meet;
      end loop;
   end Worker;

   ...
end ParallelQueens;
```

Figure 5: Parallel solution for the queens' problem. See Fig. 1 on page 3 for the procedures `implies` and `abs`. The unary operator "`#`" returns the number of elements in a compound data structure.

4.5.3 Parallel matrix multiplication

The program in Fig. 6 on page 26 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 in tuple space `RESULT` via changing `meet` operations. The counter that is contained in the tuple, which contains 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`). You could reduce the number of workers as done in [ACG86, page 30].

4.5.4 The dining philosophers problem

The dining philosophers problem was posed originally by Dijkstra [Dij71], and is often used to test the expressivity of new parallel languages. A problem description together with a C-Linda solution is given in [CG90]:

> "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. If the table is full and all philosophers simultaneously grab their left chopsticks, no right chopsticks are available and deadlock ensues." ([CG90, page 182])

The original problem specification uses a bowl of spaghetti and the utensils were forks [Dij71], 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. 7 presents the general form of a solution for the mutual exclusion problem (see e.g. [BA90, And91b]).

Some extensions to the dining philosophers problem were proposed: the drinking philosophers problem, where neighbors are allowed to drink simultaneously provided they are drinking from different bottles [CM84]. 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 [Haz91]. The problem of the restaurant for dining philosophers with multiple tables and change management [Cia91], among others. In this paper we consider the *plain* dining philosophers problem.

The solution for the dining philosophers problem in ProSet syntax in Fig. 8 is derived from the C-Linda version in [CG90]. It 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 of his left and of his right chopstick. In [CG90] this is shown with the *pigeonhole principle*. We quote for the sake of completeness:

> "We can show that this is true by using a simple combinatorial argument based on the pigeonhole principle. Consider $n-1$ bins and n balls. In every distribution of the n balls among the $n-1$ bins, there must be at least one bin that winds up holding at least two balls. Now if we associate each bin with a philosopher and each ball with a chopstick, it's clear that the set of all legal distributions of chopsticks to philosophers is a subset of the possible distributions of balls into bins — a chopstick can be grabbed only by the fellow on its right or on its left, and hence the corresponding ball can only land in one of the corresponding two bins. Since every distribution in the original set must assign two balls to

```
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 [ || 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 |(type ? = tuple) ) at WORK end meet;
      meet ( "B", j, ? column |(type ? = tuple) ) at WORK end meet;
      Dot := DotProduct (row, column);
      meet ( "C", ? |(type ? = tuple)   into AddDot(?, i, j, Dot),
                  ? |(type ? = integer) into ?+1 )
          at RESULT
      end meet;

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

Figure 6: Parallel matrix multiplication.

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

Figure 7: Form of mutual exclusion solution.

```
program DP;
   visible constant n := 5,                    -- #philosophers
                    TS := createTS(om);
begin
   for i in [ 0 .. n-1 ] do
      || phil(i);
      deposit [ "chopstick", i ] at TS end deposit;
      if i /= n-1 then
         deposit [ "room ticket" ] at TS end deposit;
      end if;
   end for;

   procedure phil (i);
   begin
      loop
         think ();
         fetch ( "room ticket" ) at TS end fetch;
         fetch ( "chopstick", i ) at TS end fetch;           -- left chopstick
         fetch ( "chopstick", (i+1) mod n ) at TS end fetch; -- right chopstick
         eat ();
         deposit [ "chopstick", i ], [ "chopstick", (i+1) mod n ],
                 [ "room ticket" ] at TS
         end deposit;
      end loop;
   end phil;
end DP;
```

Figure 8: Solution for the dining philosophers problem.

> one bin, it follows that every distribution in any *subset* of the original set must also assign two balls to one bin, and we conclude that at least one philosopher will be guaranteed to succeed in grabbing two chopstick." ([CG90, page 183])

Note that eating is considered to be a critical section which will eventually terminate and that a fair process scheduler is presupposed.

The C-Linda solution does not guarantee fairness of choice, it relies on a fair runtime system. With the guarantees given in section 4.2.4 we can proof the following for the ProSet solution:

Theorem: No individual starvation can occur, i.e. the solution is fair.

Proof: This is a very strong requirement in that it must be shown that *no* possible execution sequence of the program, no matter how improbable, can cause starvation.

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 not deadlock and thus return the room tickets after finite periods of time (general assumption for critical sections). 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 (section 4.2.4).

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. It would not be a sufficient assumption in real-time applications, where a philosopher could starve after a finite period of time of e.g. two month.

The possibility of a particular philosopher being starved to death by a conspiracy of his two neighbors, which is contained in the solution in [Dij71], is not present in our solution.

We sketch some constraints that solutions for selective mutual exclusion problems must satisfy:

- Deadlock freedom
- Fairness
- Processes that fail within their non-critical sections do not prevent others from eating. We quote form [BA90] (see also [And91b, page 98]):

  > "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." ([BA90, pages 27–28])

- 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' [BA90]
- Concurrency (non-neighboring philosophers may eat simultaneously)
- No assumptions concerning the relative speeds
- Efficient behavior under absence of contention

Historical note The solution for the dining philosophers problem, which was presented in [Has91a], did not meet the constraint that processes, which fail within their non-critical sections, do not prevent others from eating. Thanks are due to Nick Carriero and Mike Factor for pointing out this fact to me. [Haz91] presents a fair Linda solution to the dining philosophers problem. Similar to the solution in [Has91a] it is presupposed that thinking always terminates. Scott Hazelhurst writes:

> "How long a philosopher eats or thinks for is non-deterministic (but finite)." ([Haz91, page 46])
> "The code as presented does have the problem that a slow-thinking philosopher might hold up the entire table." ([Haz91, page 53])

If someone knows a C-Linda solution to the dining philosophers problem, which satisfies *all* the above-mentioned constraints, then the author would appreciate obtaining a copy.

5 Implementation issues

The compiler construction system Eli 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 [GHL+92]. A first development step using Eli is documented in [Has91b]. A prototype implementation for testing and code development on Unix workstations is under way.

The unique copy method for storing the tuple space, where each tuple is stored only once, with sending of templates seems to be reasonable on such a system for our purposes. 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, as it is done in [Wil91a]. Multiple tuple spaces allow the programmer to partition the communication device as he sees fit and simplify compile-time analysis with respect to partioning the tuple space. However, it may be reasonable to let the programmer map tuple spaces, which occur in the ProSet program, onto designated processors for a specific parallel target machine. This should be done in a separate configuration language outside the original program. In [Wil91a] this is done for programmer-specified tuple-to-store mapping functions which override the system's default mappings. As such concerns apply only to implementations on specific machines, it is not acceptable to incorporate such things into the language (portability would not be maintained). It seems to be obvious to store the tuple space in a hash-based way, since it is not necessary to guarantee fair selection of deposited tuples (section 4.2.4). The representation of individual tuple spaces may be customized according to their contents. This may be based on suitable compile-time analysis as in [Jag90].

The `meet` operation allows user-defined high-level operations on shared data (section 4.2.3). It is not necessary to move a tuple that will be changed physically from tuple space to another place for performing such an operation. It is sufficient to set a write-lock for the duration of the operation.

Templates, which belong to suspended processes because no tuple matches them, are weakly fair matched with later deposited tuples. The implementation may guarantee this via using FIFO queues for pending templates. In a FIFO (first-in first-out) queue always the possible candidate who waits longest is selected first.

Multiple tuple spaces provide a direct approach for concurrent access to tuple space: each tuple space obtains its own manager process. However, selective fetching from multiple tuple spaces may be difficult to implement with this approach.

6 Conclusions and future work

In this paper 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, specified fairness and selection for matching, 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.

The small examples presented here did not fully demonstrate the advantages of multiple tuple spaces. However, in more sophisticated problem domains such as process trellises [Fac90] the advantage of information hiding is obvious, since processes may communicate within isolated tuple spaces independent of communication in other tuple spaces. The enhanced facilities for nondeterminism will support distributed implementation of backtracking such as branch-and-bound applications, where selective waiting for multiple events is often desired [KBT89].

Future work will address the specification of formal semantics for generative communication as presented in this paper and the implementation aspects that were discussed in section 5.

References

[AB89] 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.

[Aba91] R. Abarbanel. Distributed object mangement with Linda. Technical report, Boeing Computer Services, The Boeing Company, Seattle, WA, August 1991.

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, 1986.

[ACGK88] 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.

[And91a] B.G. Anderson. *Persistent Linda – Adding Transactions to a Parallel Programming Model.* PhD thesis, Courant Institute, New York University, NY, April 1991.

[And91b] G.R. Andrews. *Concurrent Programming.* Benjamin/Cummings, 1991.

[ANP89] 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.

[AS91] B.G. Anderson and D. Shasha. Persistent Linda: Linda + Transactions + Query Processing. In D. LeMetayer, editor, *Proceedings of the Workshop on Research Directions in High-Level Parallel Programming Languages*, Mont Saint-Michel, 1991. IRISA-INRIA.

[BA90] M. Ben-Ari. *Principles of concurrent and distributed programming.* Prentice Hall, 1990.

[Bal91] H.E. Bal. A comparative study of five parallel programming languages. In *Proc. European Spring Conference on Open Distributed Systems*, Tronsø, Norway, May 1991.

[BCG89] R. Bjornson, N. Carriero, and D. Gelernter. The implementation and performance of hypercube Linda. Research Report 690, Yale University, New Haven, CT, 1989.

[BCG+91] 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.

[BHK89] 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, *Proceedings of CONPAR'88*, New York, 1989. Cambridge Univ. Press.

[BP91] P. Broadbery and K. Playford. Using object-oriented mechanisms to describe Linda. In Wilson [Wil91b], pages 14–26.

[But91] P. Butcher. Lucinda. In Wilson [Wil91b], pages 27–38.

[Car87] N. Carriero. Implementation of tuple space machines. Technical Report 567, Yale University, New Haven, CT, December 1987. PhD thesis.

[CCH91] C.J. Callsen, I. Cheng, and P.L. Hagen. The AUC C++Linda system. In Wilson [Wil91b], pages 39–73.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[CG90] N. Carriero and D. Gelernter. *How to write parallel programs.* MIT Press, 1990.

[CG91] N. Carriero and D. Gelernter. New optimization strategies for the Linda pre-compiler. In Wilson [Wil91b], pages 74–83.

[CG92] N. Carriero and D. Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, February 1992.

[Cia91] 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.

[CKM91] 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.

[CM84] K.M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, 1984.

[DF89] E.E. Doberkat and D. Fox. *Software Prototyping mit SETL.* Leitfäden und Monographien der Informatik. Teubner-Verlag, Stuttgart, 1989.

[DGH90a] E.E. Doberkat, U. Gutenbeil, and W. Hasselbring. SETL/E – A prototyping system based on sets. In W. Zorn, editor, *Proc. TOOL90*, pages 109–118. University of Karlsruhe, November 1990.

[DGH90b] E.E. Doberkat, U. Gutenbeil, and W. Hasselbring. SETL/E Sprachbeschreibung Version 0.1. Informatik-Bericht 01-90, University of Essen, March 1990.

[Dij71] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[DM90] U. Dahlen and N. MacDonald. Sheme-Linda. In Gupta [Gup90].

[Dob90] E.E. Doberkat. A proposal for integrating persistence into the prototyping language SETL/E. Informatik-Bericht 02-90, University of Essen, April 1990.

[Dou89] P. Dourish. A Transputer-based Parallel Lisp. Technical Report ECSP-TN-19, Edinburgh Concurrent Supercomputer Project, June 1989.

[Eme90] 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.

[Faa91] C. Faasen. Implementing tuple space on transputer meshes. Master's thesis, Department of Computer Science, University of the Witwatersrand, Johannesburg, South Africa, February 1991.

[Fac90] 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.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[Gel89] D. Gelernter. Multiple tuple spaces in Linda. In *Proc. Parallel Architectures and Languages Europe (PARLE'89)*, volume 2, pages 20–27. Springer-Verlag LNCS 366, June 1989.

[GHL+92] 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.

[GM88] R.P. Gabriel and J. McCarthy. Qlisp. In J.S. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*, pages 63–89. Kluwer Academic Puplishers, 1988.

[Gup90] A. Gupta, editor. *Proceedings of the Europal/BCS-PPSG Workshop on High Performance and Parallel Computing in Lisp*, Twickenham, London, UK, November 1990.

[Hal85] R.H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[Has91a] W. Hasselbring. Combining SETL/E with Linda. In Wilson [Wil91b], pages 84–99.

[Has91b] W. Hasselbring. Translating a subset of SETL/E into SETL2. Informatik-Bericht 02-91, University of Essen, January 1991.

[Haz91] S. Hazelhurst. A Linda solution to the evolving philosophers problem. *South African Computer Journal*, pages 44–53, September 1991.

[Hup90] S. Hupfer. Melinda: Linda with multiple tuple spaces. Research Report 766, Yale University, New Haven, CT, February 1990.

[Hut90] D. Hutchinson. Linda meets Lisp. In Gupta [Gup90].

[Jag90] S. Jagannathan. Semantics and analysis of first-class tuple spaces. Research Report 783, Yale University, New Haven, CT, April 1990.

[Jag91] S. Jagannathan. Customization of first-class tuple-spaces in a higher-order language. In *Proc. Parallel Architectures and Languages Europe (PARLE'91)*, volume 2 of *Springer-Verlag LNCS 506*, pages 254–276, June 1991.

[Jel90] R. Jellinghaus. Eiffel Linda: An object-oriented Linda dialect. *ACM SIGPLAN Notices*, 25(12):70–84, December 1990.

[KBT89] 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.

[KSS84] P. Kruchten, E. Schonberg, and J. Schwartz. Software prototyping using the SETL programming language. *IEEE Software*, pages 66–75, October 1984.

[Kwi89] M.Z. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7):371–386, 1989.

[Lei89] J.S. Leichter. The VAX Linda-C User's Guide. Research Report 615, Yale University, New Haven, CT, December 1989.

[Lel90] W. Leler. Linda meets Unix. *IEEE Computer*, 23(2):43–54, 1990.

[Luc86] S. Lucco. A heuristic Linda kernel for hypercube multiprocessors. In *Proceedings of the SIAM Conference on Hypercube Multiprocessors*, September 1986.

[Mac91] N. MacDonald. Linda work in Edinburgh. In Wilson [Wil91b], pages 100–104.

[MK88] S. Matsuoka and S. Kawai. Using tuple space communication in distributed object-oriented languages. In *Proceedings of OOPSLA'88*, pages 274–284, San Diego, September 1988.

[Par90] H.A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.

[Sch91] G. Schoinas. Linda and the Blackboard Model. In Wilson [Wil91b], pages 105–116.

[SDDS86] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets – An Introduction to SETL*. Graduate Texts in Computer Science. Springer-Verlag, 1986.

[She90] A.H. Sherman. *C-Linda Reference Manual*. Scientific Computing Associates, New Haven, CT, 1990.

[SP90] G. Sutcliffe and J. Pinakis. PROLOG-Linda: An embedding of Linda in muPROLOG. In *Proceedings of AI'90 – the 4th Australian Conf. on Artificial Intelligence*, pages 331–340, Perth, Australia, 1990. World Scientific, Singapore.

[Wil91a] G. Wilson. Improving the performance of generative communication systems by using application-specific mapping functions. In [Wil91b], pages 129–142.

[Wil91b] G. Wilson, editor. *Linda-Like Systems and Their Implementation*. Edinburgh Parallel Computing Centre TR91-13, June 1991.

[WL88] R.A. Whiteside and J.S. Leichter. Using Linda for supercomputing on a local area network. In *Proceedings of Supercomputing*, pages 192–199, Orlando, Fla., November 1988.

[YW90] C.K. Yuen and W.F. Wong. A self interpreter for BaLinda Lisp (Version 0.1). Technical Report TRD2/90, DISCS, National University of Singapore, February 1990.

[Zen90] S.E. Zenith. Linda coordination language; subsystem kernel architecture (on transputers). Research Report 794, Yale University, New Haven, CT, May 1990.

Index